2

Reliability Techniques for Computer Executive Programs

Summary Report

NAS8-2666-9

CR-123736

Information Research Associates

2200 San Antonio

Austin, Texas 78705

May 17, 1972

1-a

1. Introduction

The study of techniques for increasing the stability and reliability of computer programs is still in its infancy. This report deals particularly with those techniques especially adopted for use on executive and supervisory systems. This summary report is in three sections, Program Structure, Program Validation and Redundancy Techniques: Roll-Back and Recovery. Each report section corresponds fairly closely to the section in the scope of work specification.

2. Program Structure

Contemporary program segmentation as manifested by general explicit control structure, phase segmented code, page organized programs, parallel fragmented tasks, and rollback structured routines are assessed for reliability characteristics. All structures enhance the program's reliability by ordering the code bulk into smaller interacting components and defining the interface characteristics. Segmentation criteria provide basic information for the confirmation and validation activities.

The explicit and phase structures are natural outgrowths of the design procedure requiring minimal achievement cost. Page, parallel, and rollback structures require substantial processing investments to obtain the best partitioning; their reliability impacts are more applicable to later phases of check-out when the code becomes fairly stationary.

The page and rollback structures demonstrate inherent hardware failure recovery mechanisms. Parallel structure allows for graceful degradation in the hardware environment.

To improve confidence and reduce validation lead time, several actions may be taken. Implementation primitives (e.g. hardware microprogrammed or software interpretive procedures) can be designed for diagnostic goals to exercise and validate new programs. Streamlined primitives for application efficiency would be substituted without modifying the program code. An automated system is desirable and feasible to assist in the checking (both dynamic and static) of new programs. Emphasis would be placed on interface consistency checking among cooperating sections and automation of tedious debug procedures to speed programmer analysis of operational results.

3. Program Validation

Five techniques for validation of programs are discussed. The effectiveness of each depends to a marked extent on the modularization techniques defined in the allowed program structure.

3.1 Automatic validation of modules through rigorous proofs of correctness of the programmed representation of the algorithms (see Appendix A).

3.2 Automatic validation by exhaustive or selective analysis of all flowpaths through selected modules.

3.3 Analysis of validity of data bases used by the modules. This can be accomplished by examination of explicitly transferred parameters and by automatic analysis of access and sequencing mechanisms for global common data bases (see Appendix B).

3.4 Rigorous modularization allows the use of mixed execution-simulation testing. That is, program modules may be replaced in the testing phase by simulation of their functions. This simulation may include such features as generation of limiting test cases for calling and called modules and extensive validity checking for transmitted data. Such simulation conveniently allows total environmental testing of only partially coded total programs through the simulation and use of data structures which are not yet implemented.

3.5 A further possibility whose convenience is enhanced by rigorous "top down" modularization is mixed compilation-execution and interpretation of source code.

Economic means of construction of an integrated validation package including all of these components are discussed.

4. Redundancy Techniques: Roll-Back and Recovery

Reliability is an important aspect of any system. On-line diagnosis, parity check coding, triple modular redundancy and other methods have been used to improve the reliability of computing systems. In this paper another aspect of reliable computing systems is explored. The problem is that of recovering error-free information when an error is detected at some stage in the processing of a program.

If an error or fault is detected while a program is being processed and if it cannot be corrected immediately, it may be necessary to run the entire program again. The time spent in rerunning the program may be substantial and in some real time applications critical. Recovery time can be reduced by saving states of the program (all in the information stored in registers, primary and secondary storage etc.) at intervals, as the processing continues. If an error is detected program is restarted from its most recently saved state. However, a price is paid in saving a state in the form of time spent storing all the relevant information in secondary storage. Hence it is expensive to save the state of the program too often. Not saving any state of the program may cause an unacceptable large recovery time. The problem that we solve is: Determine the optimum points at which the state of the program should be stored to recover after any malfunction.

4-a

Reliability Techniques for Computer Executive Programs

Final Report

NAS8-26669

Information Research Associates

2200 San Antonio

Austin, Texas 78705

May 17, 1972

3-a

TABLE OF CONTENTS

I. PROGRAM STRUCTURE

Program segmentation is the process of classifying general program elements into collectively equivalent component categories. Properties depend upon the decomposition criteria used. For example, most program parts are naturally segmented into data and instructions (except for self-modifying code). Data may be hierarchily refined into files, records, words, and bits of information when this structure is desirable for exemplifying or coding the program. Similarly, instructions may be functionally grouped into nested or cascaded routines, specifying their interaction and cooperation in the software system.

In this section we will be primarily interested in exposing segmentation characteristics in program instructions and data. The programs may be fragmented for operational clarity, debugging assistance, validation and evaluation probe insertion, more amenable hardware operation, or recovery from transient faults. Since many considerations aside from reliability influence the final segmentation selected, it appears prudent to examine the characteristics of some segmentation schemes and observe how they enhance or mitigate diagnostic efforts.

The program graph.

For examination and description, it is often convenient to represent the program as a directed graph. In this abstract representation, a collection of nodes are interconnected by directed edges. The relationship between the program and its graph may be manifested in several ways depending upon the analysis intent. For example, nodes in the graph may correspond to clusters of instructions and edges describe the sequence of control among these actions (e.g. a flowchart). Alternatively, the nodes may represent instruction sequences and the edges associated with data linking the operations together. Since the first interpretation is more widespread, we will assume this form here.

Graphical representation allows considerable flexibility for program analysis. For example, we may choose to group several nodes and their interconnecting edges together and attribute a processing property to this subgraph. After making desirable groupings in the original graph, we can construct a new graph consisting of nodes corresponding to previous subgraphs and edges connecting the subgraphs. By condensation of the program graph, we can illustrate the process in any level of detail desired.

In some applications of segmentation, it will be desirable to produce modified graphs that are loop free. This acyclic graphical representation insures that once a node's processing is complete, the system never reenters this node. To perform the reduction, loops in the original graph are coalesced to nodes. If looping is complex or there are nested loops, it may be desirable to have several intermediate levels in the description between the original graph and its acyclic reduction. If this coalesce oversimplifies the structure, it may be necessary to remove some feedback edges (i.e. those causing looping) to obtain a cycle free graph. In general, the minimum number of edges is removed.

It is obvious that the coagulation of nodes segments the program into parts. Grouping nodes of the original program graph nodes defines segment members or program tasks. Since loops typically indicate itterative procedures or set operations, defining segments along loop nesting lines is reasonable.

Types of segmentation

Since most programs are segmented to some extent by either the language or design process while they are developed, it will be extremely infrequent that one encountered software completely void of structure. Therefore, we will examine the nature and characteristics of this prior segmentation and observe how it may be used for reliability considerations.

The most frequently encountered type of segmentation is produced by fragmenting the process into phases. While the decomposition is often performed manually during program design, it can be introduced after the fact by producing an acyclic program graph. The system operates by sequential application of phases which are process oriented and communicate by common data and directed calls to the next phases. In operation, the phases may use memory overlays to conserve space. For example an assembler might be segmented into first and second pass phases where the first pass builds a symbol table, assigns addresses, and expands macros that are used by the second pass phase in producing object code.

Other program structures may have been imposed to accomidate operation on specific hardware. In this category, the system may be decomposed into serial self-sufficient tasks to run simultaneously. This parallel segmentation allows reduced completion time when several hardware processors are potentially available or considered separate in a multiprogram environment.

While multiprogrammed systems may be manifested in many forms, the two most prevalent are batch and paged systems. The batch system requires entirely main memory resident programs with complete jobs swapped in and out as required by I/O requests or system functions. Since pure batch processing does not affect a program's structural properties significantly, we will not direct attention to it. In paged systems, however, only a small portion of the program is present in main memory at any given time. The philosophy is that at any particular instant, the majority of processing is local—instructions in the vicinity of the current instruction are more probable than remote instructions and data references tend to be localized. That is, each section of the program can be characterized by a "working set" of instructions and data that are used most heavily. Program references (instruction or data) outside the current working set (called a "page fault") indicates a change in execution characteristics and requires a new (possible overlapping) working set. Thus the program is organized into pages such that a set of pages corresponds to the working set required for significant amounts of processing. Attempts are made to place loops on the minimum number of pages to reduce the number of page swaps. Similarly, the data areas required for the looping segment are grouped on the same data page. The program operates by having the program information on mass sotrage and placing in main memory only the working set required at any instant. Page faults cause the old pages of the working set to be replaced by the required pages in main memory. Old pages are updated on the mass storage copy to maintain execution changes and reflect the state of the process when swapped. Even when not a formal paging system, many similar characteristics exist when instructions are multiply fetched or a cache memory system exists.

Rollback program structure is directed primarily toward diagnosis and improved reliability. At various program points (rollback points) the program state is observed and recorded. Execution then commences to the next rollback point.where results are examined for correctness. If correct, the program state is updated and execution resumes on the next section. If an error is detected, the program state is returned to the last rollback point and the program segment in error is rerun. This scheme is intended to recover from transient hardware errors and prevent solid faults from permanently damaging the data base (e.g. files used as both inputs and outputs). The optimal location of the rollback points will depend upon the overhead cost of saving the program state, and minimizing the cost of lost processing time. In part, the location of these points will require locating the places in the program in which the minimum amount of information describes the program state. We will assume here that the error detection criteria can be specified and the rollback procedure itself is error free.

Notice that the different segmentation schemes are not necessarily mutually exclusive. For example, the software system may have identifiable phases which are individually composed of parallel tasks. Each of these tasks may be realized as a paged or rollback structure. Since the properties of such a hybrid system can be largely derived by the superposition of individual characteristics from the various philosophies, we will deal here with the properties of pure segmented systems along one of the themes.

Debugging ease.

Debugging a program is really an ongoing process rather than an instantaneous development task. Ordinarily, the procedure is renamed "maintenance" after the program is released for use even though actual implementation errors are corrected. In any case, we can identify at least two classes of activity: 1) initial operational investigation and 2) advanced inspection of suspicious events. The first activity involves the removal of keypunch errors and bad coding to the point that the program compiles and successfully executes test cases.

To realize these goals, the programmer usually employs two tactics:

1) Imbedded diagnostic print commands

2) System dumps of the program

Effective utilization of diagnostic print commands requires programmer precognition of both the program architecture and probable causes of errors. Failure to anticipate these problems results in the insertion of new print commands, recompilation, and additional run time to repeat the failure; obviously, duplication of known errors is poor utilization of both computational and human resources. Offsetting advantages are the ability to capture transient behaviors, exact interpretation of bit patterns (e.g. decoded to floating point), and concentration on meaningful subsets of information.

System dumps are provided either on demand or as a post-mortem for detected fatal errors (e.g. expired time limit, memory protection violations, etc.). The completeness and format of the information, however, often impedes the investigation probing. For example, to determine the value of a single program variable, the programmer must compute the physical address using a symbol table with relative addresses and several relocation biases. Faithful execution of this exercise is rewarded with a compressed binary pattern (e.g. octal or hexadecimal) requiring further decoding to the desired representation (e.g. decimal integer or floating point). This process is both tedious and error prone--largely because the information is organized for convenient implementation in the computer. Since most of the required information is already available, interactive automated assistance is both valuable and tractable. The user could symbolically request variables and specify the decode format. With access to cross reference, symbol table, and source code from the compilation process, the interactive system could be used to present lists of interacting variables and their values from the dump data. For example, consider the FORTRAN code of Fig.1a and directed graph representation in Fig. 1b. Suppose the programmer determined that the value for F was erroneous and was searching for the cause. Beginning in the source code at (*), the automated system could develop the relationship of Fig. 1c, showing the input variables contributing to the value of F. The values of these variables, decoded in the format implied by the source code, would be presented to the programmer for analysis. If a subset were found in error, the system would continue to track those paths until the problem was found. The amount of data generated could be regulated by an upper bound on the number of variables or depth in the relationship structure.

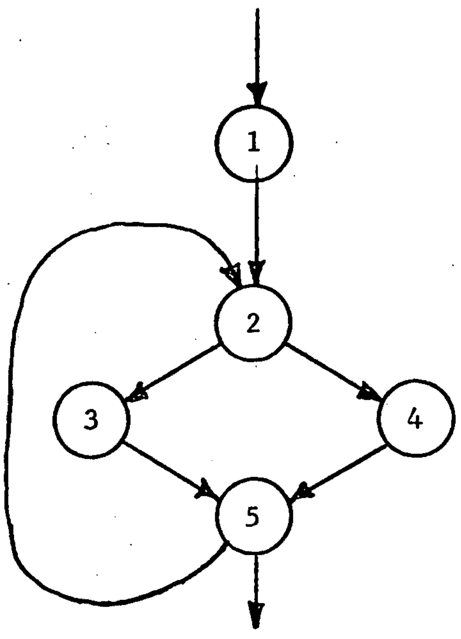
The interactive system would remove the tedious mechanisms and provide a complete list of inputs (some, quite obscure) to the area under investigation. The decode format indicates typing assumed by the source code which may have been overlooked by the programmer. Many troublesome indirect relationships can be quickly discovered in the interactive investigation which might be elusive in the manual equivalent.

When reasonable confidence exists that the program is correct for test cases and probably will not mutilate itself or the system, it is released for use, and testing enters the second phase. In this second effort, anomalies observed by users are examined and the sources of difficulty discovered. As changes to the system are made to correct malfunctions, new

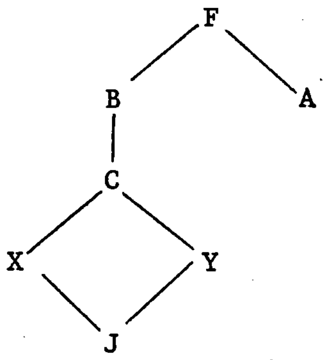
7 (a)

```
              R = 3                                  }
              A = 100.0                              }
              Y = 1.0                                }   1
              X = 2.0                                }
              Q = X**2                               }

              DO  500  J = 1,100,3                   }
              S = F**3 - X*Y                         }   2
              C = X**2 - 2*X*Y + Y**2                }
              IF (C.LE.3) GO TO 100                  }

              B = C-3                                }
              P = Q**2                               }
      *       F = ABS(B**2) - SQRT(A)                }   3
              P = P+1                                }
              X = X-J                                }
              GO TO 500                              }

   100        C = ABS(C-3)                           }
              Y = Y + J*2                            }   4
              Q = Q*R+Y                              }

   500        CONTINUE                               }   5
```

(a)



(b)



(c)

Fig.1

errors may appear as other portions react to the change and errors in the modification are corrected. Hopefully, the process converges and eventually all errors are purged from the program.

Whereas the first type of debugging is characterized by catastrophic failures and complete lockup in the program, the second typically demonstrates sensitivity to a particular situation and partial malfunction in the operation. In initial debugging,close control is required and very detailed tracing of program operation is performed to insure that individual small sections of code are properly coded and translated. After these low level operations are verified, the program is allowed to run more freely and details of the system examined for specific types of anomalies. Usually, this level requires dumps of the system and determining what the process was doing when the error occured. The suspected error may be tested for repeatability and independence from preceeding events.

Serial segmentation into phases allows some independence in the total system that isolates large segments from one another in the debugging venture; these segments, however, tend to be quite large and may not offer significant advantages in the debugging effort. Conversely, the paging segmentation organization may not provide lucid properties to the debugger. It does offer the advantage that interacting data is typically located on the same page. It is not, however, worthwhile to expend the processing time to organize pages if the program is in a state of rapid change. Therefore, the advantages of a page organized program will probably become useful in advanced stages of debugging rather then at the onset. Similarly, the information contained in segmenting for parallel processing and insights into the cooperation of program subparts obtained will be more useful in advanced debugging than the initial phases.

Rollback may actually increase the debugging effort if the programmer is required to code validity tests at rollback points. During initial debugging, the designer may not have a good feel for operational qualities of the program. Therefore, the production of the tests at rollback points may force the programmer to analyze the system more extensively than he would otherwise, with salutary effects. The production of the rollback tests may be a fruitful middle activity between the initial checkout and more advanced stages of certification.

Hardware Failure Tolerance

Analysis of program tolerance to hardware failure encompasses two aspects:

a) What is the minimum amount of available hardware the program must have to run effectively?

b) How sensitive is the program to the failure of a particular hardware component?

We can further classify failures into hard faults and transient faults; either of these types may produce loss of program control or errors in data manipulation. Of these various types of failure, the transient error resulting in temporary loss of control or imperfect data is the most insidious with high potential for nonrepeatable errors that may go undetected for long periods of time. Hard faults resulting in significant malfunction are readily identifiable and can be combatted with proper backup restart procedures after a minimum hardware configuration has been reestablished.

Phase segmented programs are the most straightforward representation and require little additional hardware for proper operation. Insensitivity to hardware errors can be introduced by complete duplication at considerable expense. Transient faults may be detected at the end of a phase, however, the permanent damage may have resulted to the data base, requiring a complete restart procedure from the backup.

Paged systems require more operational equipment to perform page swapping. It can provide better transient fault detection by detecting suspicious page faults. From correctly confirmed runs, it is possible to associate with each page lists of legitimate page fault locations. If a transient error causes loss of control, the swap system could detect the malfunction from this list and take appropriate corrective action. Undetected errors would occur locally in a particular working set, hopefully with constrained effects.

Programs in parallel segmented parts are designed to run on multiple processors implying that more hardware may be required for maximum time performance. This construction, however, allows for considerable flexibility for graceful degradation of the system with hardware faults since any two segments on different processors in parallel may be run serially on either processor with no detrimental logical effects. Furthermore, since the parallel segments run independently, faults from one segment do not affect its parallel companions on other processors; errors are localized.

Rollback systems are designed to be particularly valuable for detection of faults. They require more hardware for saving states but can recover successfully from such troublesome errors as transient loss of control. They further protect against permanent destruction of files by examining for correctness before updating the program state. When errors are detected, the program always returns to a known good state for retrial.

Sensitivity to Software Logic Errors

As previously described in the section on debugging, the discovery of logic errors fall into categories of initial errors and advanced inspections. Since initial errors are best discovered by straight forward segmentation and the inadvisability of investing computational resources for sophisticated segmentation schemes on questionably stable programs, we will deal here with the problems associated with advanced logic errors (i.e. "maintenance" problems).

The single most valuable tool in combatting software logic errors is prevention. To this end, all the segmentation schemes provide insight into the system interconnections enabling the programmer to recognize how tentative modifications will be received. Armed with this information, collectively acceptable changes can be tailored. Paged organizations provide some information on variables affected since interacting quantities should have common pages. Parallel organization will indicate the precedence relations among the components. Similarly, phase oriented organizations will provide the interconnection of large components with each other. In all these cases, however, immediate neighbor information is more easily obtained, while complete interconnection must be obtained by transitive relationships among the adjacent portions.

After appropriate preventative measures have been applied, the detection of persisting errors can be performed during trial runs of the modified version. In paged systems, the allowable page transitions previously collected can be used to indicate where changes have modified this operation. This data will assist in spotlighting suspicious operational characteristics that can be confirmed as correct or identified as errors.

If, after modification, the parallel segmentation analysis is performed, the precedence relationships can be compared between the previous version and the new one. Changes in dependency among the tasks should be investigated and confirmed. Furthermore, the analysis will establish the precedence of new code to be confirmed with the hypothesized dependency.

Checking Values and Times

To verify the proper operation of the program, it is desirable to examine dynamically the values of significant data. Furthermore, timing investigation may provide meaningful information concerning the code integrity of a section. The number of inspection interfaces and their location depends in part on our confidence in the code section and how closely we wish to monitor the operation. If too many points are monitored, the inspection overhead may degrade the program's operation and produce distorted timing results. Furthermore, the very mass of data generated may obscure overall operational considerations. If the program is not monitored frequently enough, checkpoints may not be reached by nonterminating program segments. By the time the malfunction is discovered, the resulting system state may be so mutilated that it is impossible to decide the cause.

With currently available techniques it is possible to catalogue all variables used in the program and automate the insertion of the software probes at all places a variables is used. Thus, once significant variables have been selected, the investigator is relieved of the task of locating all occurrences and must simply code the appropriate check for the selected variable. This checking may take the form of absolute bounds, conditional situation dependent upon the dynamic value of other variables, or a particular type of sequential activity (e.g. monotone increase by fixed increment) possibly identified with a particular program section. The investigator may choose to limit the inspection to only those activities which modify the variable value (i.e. where it is an output of some activity), only where it affects results (i.e. where it appears as an input), or only in a particular section of the code (abridging the complete list of variable activity pointers). Thus the investigator has the capability to specify the level of detail desired in the monitoring by selecting the data to be observed.

The examination of timing information may require collecting frequency of a segment's execution or the aggregrate time expended on a section. The first aspect is more easily implemented while the second provides additional details. This timing data could be used to improve the program's performance or to study its operational variation to demand. Actual time information appears more valuable in realtime applications and where an unduly heavy or light computational effort appears contrary to expectations.

All the segmentation schemes provide to some extent the recognition of local variables used only for intermediate and short term results. Major control data and variables of most significance is usually more globally defined. If the program is segmented in phases, the size of the segments may be quite large. Inspections may be desired more frequently than the phases change.

Although paged segmentation does not directly assign pages along logical program lines, the reduction of page swaps and working sets tend to correspond more to the program logic than a random selection. The checking of values and time at page swap times would allow considerable autonomous processing to occur before examining results; permanent changes to the program state in mass storage may be conditional on the checking outcome. Similarly, the parallel processing segmentation provides convenient and meaningful points of investigation in the interim between the completion of one task and the scheduling of the next.

The rollback segmentation is designed specifically to optimize the checking of interim results before advancing the program state. This type of structure is predicated on minimum checking within acceptable recovery and examination limits. Thus it is the most logical structural type for this type of activity. In fact, the examination of acceptable results is critical to rollback operation since this analysis forms the basis of decisions to proceed or retry.

Independence and Integration of Program Segments

Since all segmentation schemes by definition provide isolation from neighboring code, they are all partially valuable for the independent checkout of logically separate functions. Some of the techniques, however, provide more reasonable structure than others. For example, the page organized structure may not provide exactly the type of separation required for stand alone checkout. Furthermore, in integration, the organization of code on pages may change considerably from the form when analyzed for stand alone operation.

Parallel structures, however, will largely maintain their precedence relationships when integrated. Some of the tasks may become dependent or establish a requirement for tasks external to the stand alone function, but the relationships among the tasks within the segment maintain relationships relative to each other. Integration then requires relating boundary tasks dependency between two subparts, preserving computational investments in segmenting the subpart.

The most logical candidate for independent checkout is the phase oriented structure where the parts of the program are identified with distinct functions and thus quasi-independent in nature. Integration of these independently verified portions usually requires manual effort since any interface requirement may exist. This problem may be minimized by strict definition of the communication structure between program subparts and verifying compatible specifications between originator and user segments.

Rollback structures provide the necessary mechanisms for restarting the program from the last rollback point. Therefore, its value is largely in assisting with the debugging of the integrated system by concentrating processing power on the malfunctioning area rather than running preceding sections to condition the program state for the suspect segment.

Instruction Set and Linkage Effects on Reliability.

Since the instruction set and linkage characteristics enhance or impede translation efforts from abstract concepts into machine compatible form, their selection impacts the software reliability. Inconsistent conventions, ambiguous operations, and poorly defined operational specifications contribute to coding errors. Conversely, concise and logically compatible instructions greatly reduce misinterpretations or hardware functions. Establishing common ground of understanding between hardware operation and the programmer can take two forms. The classical approach is through operational documentation and providing instructions useful to collective target user classes. Alternatively, a machine with restructurable qualities allows users to construct languages and conventions they desire. Selecting a suitable instruction repertoire for the process can significantly reduce code bulk, limiting opportunities for keypunch errors and providing organizational clarity.

In accommodating the user, the hardware usually assumes some additional burdens. For example, the information storage might include typing information using tag bits as well as value information. Thus, generalized memory words become executable instructions, integer data, floating point numbers, logical values, array members, etc. The instruction set is reduced since argument contextual information dynamically augments the instruction op code removing the necessity of type explicit mnemonics (e.g. Integer ADD, Half word ADD, Floating Point ADD, Floating point double precision ADD, Logical ADD, etc. become simply ADD).

Similarly, special operations may be included to accommodate operations required by segmentation schemes. In the coordination of parallel tasks, a "test and set" instruction allows easy lock and key operations. With this operation, one can simultaneously test for availability, obtain access, and exclude other processors without resorting to elaborate software routines (e.g. queued requests serviced by a master scheduler). Rollback implementation is facilitated by instructions designed for examining program state attributes, and allowing easy reloading of a previous state when errors are detected. In paged systems, relative addressing is essential to efficient operation. Program pages are not necessarily loaded in the same absolute location when swapped; this dynamic residence is accommodated by loading a "base register" or performing page relative operations. The relative addressing scheme may play an important role in detecting page faults (e.g. an address overflow of the page boundary).

The desirability of tailoring descriptive constructs for a particular purpose has long been recognized. This realization has resulted in the development of macro assemblers, procedure oriented languages, and easy subroutine specification. While these techniques may offer the required degree of flexibility, the application of pyramided constructions seriously degrades operation performance compared to systems written in the computer's native tongue. With the advent of dynamically microprogrammable computer hardware, the potential exists to create instructions of comparable (and often improved) execution properties. Since most programs utilize only a small fraction of the sanctioned op codes, the user may choose to substitute a custom instruction for one he is not using. Similarly, he can modify existing conventions for compatability with the program's intent.

It should be noted that questions of instruction suitability is not a static property since operational applicability changes as the program is debugged. In the initial phases of debugging, the user might want a highly investigative operations with close examination of data types for compatible properties (e.g. identification of mixed mode operations) or possible tagged data for trace purposes at the temporary expense of value precision and range. Once validation has advanced and the program is placed in production, the investigative procedure may be removed to streamline process execution; that is, diagnostics become overhead after detectable errors are purged. If the program requires significant maintenance, the diagnostic instruction set could be reinstalled to assist integrity verification of the modified version and assure acceptance by unchanged sections.

Of course, an equivalent software test could be placed in the code to perform checks with conventional instructions. Among the advantages of instruction modification are speed and the stationary property of the software code. Since the same code is used for both test and application, segmentation schemes are not affected. Placing software checks in the code may affect the program's segmentation into pages or parallel tasks. When the checks are removed, the program must be analyzed for segmentation.

It is possible to err in defining instructions with microcode just as in any other specification, but mistakes are likely to be more pronounced by misformulated instructions. Since the instructions are such frequently used building blocks, errors here produce more.apparent malfunctions, easily detected and repaired. Investigators would be less likely to encounter subtle faults which might go undetected.

Diagnostic and protective hardware features can be powerful tools in the verification process. Such tactics as protected memory areas (e.g. unprotected, read only, and execute only) are useful in preventing programs from progressive mutilation that obscures error location. When the malfunction effects (e.g. a location improperly modified) are known and the cause is undecidable, a "break point" feature which monitors references to the symptom area may provide the fastest method of error location.

One of the greatest causes of program malfunctions is the linkage of data to subfunctions. The linkage may be provided dynamically in execution by formal parameters or compiled linkage using macros or common statements. Problem sources include misunderstanding of the subroutine operation or side effects from the execution. Since few languages require direct specification of variables classified as inputs or outputs, the problem is compounded for the user. One possible protection is assumption of all variables being subject to change; only local copies of the required data are coupled. If a parameter is modified in the subfunction, the results are ignored by the calling routine. This scheme, however, is only applicable for formal parameter linkage.

Another alternative which rectifies several problems again involves the use of in memory tagging. Since the user and subfunction each have local notions of inputs and outputs, they might independently specify tag conditions for each class. The subfunction would be responsible for checking the tags for appropriate typing. This scheme could further reduce problems of unequal parameter list lengths and respond with a warning when the condition was detected. Similarly, if tags additionally reflect data typing, the subfunction could confirm coupled data of consistent format. As before, when detected errors have been rectified, the checking overhead might be removed.

Repetitive Use of Reliable Modules

When a frequently used activity is identified, it is typically isolated as a subfunction and takes the form of a subroutine, procedure, or macro. The service subfunction is invoked by program sections requiring the activity. This allows economy in coding effort, possible reduction on memory requirements, and increased confidence of faithful operation. Clearly, it is superior to recoding the operational sequence each time since this would increase chances of transcription errors.

The subfunction manifestation will depend upon its functional characteristics. We can categorizc these characteristics as reentrant, serially reusable, and unrestricted.

Reentrant properties require pure procedure subfunction specification; any modifications to data are made relative to the invoking call. Thus, temporary values within the procedure itself are forbidden.

Serially reusable code allows temporary modification in the subfunction provided the original state is restored before exit. Thus, on a serially used basis, the subfunction is identical on each successive application.

Unrestricted code may take any form and may perform differently depending upon when it is used and how many previous executions have occurred. Although the simplest form to code, efficiency in code production may be offset by increased debugging time.

Suppose in a correctly functioning program, there is a subroutine with formal parameters. If the subroutine is unrestricted in form, we must retain a single central copy of the procedure which is shared by all invoking calls. If the code is serially reusable or reentrant, we have the option of either a single copy or replacing calls with a macro expansion of the subroutine and compiled linkage for a particular call.

If the program is parallel task oriented and subfunctions are reentrant, a single copy or expanded macro are equivalent. If the subfunction is serially reusable, a central subroutine requires lock and key protection in a multiprocessor installation. Common usage of single copy, nonreentrant subroutines establish a precedence relation among tasks potentially parallel otherwise. To remove this restriction local copies could be provided for each invoking task unless the subroutine is unrestricted. Unrestricted subfunctions require central copies with lock and key protection.

In page segmented programs, calls to subroutines may produce undesirable page swaps. Hence, restricted subfunctions expanded as macros would materially enhance the operation. Little study has been performed on the time decrease, space increase tradeoffs of this alternative.

In rollback segmented systems, central subroutines impose a restriction on roll back point placement. If the analysis determines an optimal roll back point for a program segment in the center of a subroutine, clearly this point is not necessarily optimal for other subfunction calls. Here, again, the option of expanding like a macro would be potentially advantageous.

Language Processor Reliability Role

The language processor will bear considerable responsibility for the automating analysis required to ease control and validation. Of course, it is a necessary prerequisite that this processing be highly reliable to avoid introducing errors not intrinsic in the program specification. Since bootstrapping validation process requires a firm basis, the language processor will require considerable attention; anomalies must be quickly and effectively repaired.

The language processor will be required to catalogue and correlate the symbolic program realm to the machine compatible realization. Here the variables and their use type are recorded, the program graph is developed. From this information, it is possible for segment producing programs (e.g. parallel task recognition, fitting code to pages, linkage between phases, etc.) to operate. Attention should also be given the language translation properties such that the programmer's intent and control specification may be lucidly displayed.

Toward this end, the language should allow typing distinction where linkage to unfamiliar portions is performed. In subroutine calls, the calling parameters could be distinguished as to input and output usage with declarations of nonvolatile data.

The language processor should be capable of automating simple symbolic relabeling to facilitate common nomenclature among the several components. For example, if several segments utilize the same data, the same symbolic name should be used in both segments. This requirement is difficult to realize in the initial stages of development since the common name set is unstable until the program has advanced in checkout. Furthermore, ex post facto nomenclature may conflict with a current name in the segment, creating problems in the relabeling procedure. Therefore, there must be coalescing hierarchy such that more global variables are given nomenclature preference for common linkage, local name conflicts can be automatically and completely adjusted. Automated relabeling techniques are the only modification method which appears to provide confidence of complete detection.

Some language aspects that create barriers to relabeling are the linkage by common and equivalence statements. Equivalence statements offer no real difficulty when used for alternate symbolic names with the same structure. When they are used to establish spatial relationships among structural data (e.g. arrays), the mapping of old labels to new may be more complex. For example, let there be two one dimensional arrays of size 9 and 10 items called "A" and "B" respectively. With an equivalence statement that the first element of B is identified with the second element of A, we have established that the two arrays are skewed relative to one another. If the relabeling procedure requires A to be relabeled as "TIMES" for symbolic system consistency, then references to B must be related to entries of TIMES. In this example, the transformation is fairly obvious in replacing appearances of B(I) with TIMES(I+1); in other situations, however, the changes may not be so straight forward; even if practical, the transformed representation may not provide more clarity.

Common data offers the potential for reorganizing data passed between routines and the relabeling of quantities in the common data base. For example, data represented as integers in one program may be used collectively in another routine as array elements (e.g. A, B, C, link with elements of COSTS(3) in another). In fact, this type of usage may establish a hierarchy of data structure in the program where one routine is concerned with the data on a higher level than its service subfunctions. When the variables are typed by the language (either explicitly or implicitly), the typing specified by each routine using the common data should be compared for consistency. This examination would indicate occasions of misunderstandings in multiauthor systems or when modifications are applied and serve to spotlight places where "tricky methods" restructure the data. This automated examination should include checks for the following:

1) Change of variable type
2) Scalars broken out of arrays
3) Array structure change (e.g. change in number or size of dimensions)
4) Misaligned array boundaries (e.g. where arrays do not fully cover one another between specifications)

The language process can be of significant value in the initial debugging and correction effort, particularly when highly interactive systems are available, by utilizing techniques of incremental compilation. In this system, the program is structured on a language statement level with each statement compiled directly and interconnected by an execution time supervisor. The supervisor directs the execution of the independent code corresponding to the sequence of statements, regaining control after each statement has been performed. This allows the user to step through the program or specify noninitial starting points that will concentrate on the particular problem being investigated. Data cooperation is managed by the supervisor using a retained symbol table to facilitate inquiries on a symbolic basis. This structure allows modifications to be made on a statement basis and only newly introduced code need be compiled before run results are produced.

II. PROGRAM VALIDATION

1. Introduction

The concept of program validation by exhaustive testing of all flow paths through large programming systems together with their input and output relations is an intractable task for interesting programs. Therefore, all successful efforts at program validation must begin with specification and utilization of program modularization and sub-unit structuring (some of the related concepts are explored in the section entitled "Program Structures"). It is the usual case that modularization in the design and specification process proceeds from the "top down" while the actual coding and debugging is carried out from the "bottom up". This implies that it is necessary in the process of program validation to reconstruct elaborately many local environments for program modules at the bottom of the design specification tree. We specify in this report a program validation system which is designed to enable the retention to a considerable extent of the natural "top down" outlook where this is desirable in the coding and validation phase as well as in the design specification stage.

The characteristics of modular design appropriate for rigorous program validation may be different from that desirable for modularization to produce convenience and speed in the actual coding process.

1.1 Modular boundaries must be as rigorous as possible.

1.2 Multiple entries and exits, and transfers to incompletely specified labels and program structures must be avoided.

1.3 Shared data bases, whether explicitly transmitted parameters or global tables, arrays, and other structures, should have restricted and explicit access mechanisms.

1.4 The validity of language defined characteristics of explicitly transferred data should be rigorously examined.

2. Techniques for Validation of Modular Programs.

Adherence throughout the modularization scheme of these properties allows the development of a validation system based on the following component techniques.

2.1 Automatic validation of modules through rigorous proofs of correctness of the programmed representation of the algorithms (See Appendix A)

2.2 Automatic validation by exhaustive or selective analysis of all flowpaths through selected modules.

2.3 Analysis of validity of data bases used by the modules. This can be accomplished by examination of explicitly transferred parameters and by automatic analysis of access and sequencing mechanisms for global common data bases (See Appendix B.).

2.4 Rigorous modularization allows the use of mixed execution-simulation testing. That is, program modules may be replaced in the testing phase by simulation of their functions. This simulation may include such features as generation of limiting test cases for calling and called modules and extensive validity checking for transmitted data. Such simulation conveniently allows total environmental testing of only partially coded total programs through the simulation and use of data structures which are not yet implemented.

2.5 A further possibility whose convenience is enhanced by rigorous "top down" modularization is mixed compilation-execution and interpretation of source code.

3. An Integrated Validation System

It is highly desirable to have an integrated programming (and hardware) system reflecting this conceptual structure. This suggests that ideally there would be produced a programming system with a compiler and interpreter for a problem source language and a compiler for an appropriate problem simulation language, all operating under a compatible control system. This would clearly be an elaborate and expensive system. Indeed, the production of such a system without the existence of errors in the system itself is unlikely. A more economical and reliable alternative is to develop a pre-processor system for a standard manufacturer supplied programming language. Such a pre-processor would process language forms appropriate for the problem area and to the simulation of the problem area. The translator would be, however, to the standard high level language. It would be capable of imposing coding conventions for the standard language which would imply and enforce rigorous modularization, checking of explicitly transmitted parameters, and translation checking and validation of global data access. It could also generate the environment necessary to allow calls to simulation modules or to invoke an interpreter system for modules. It could, in addition, conveniently provide a partial execution trace by imbedding dump and trace activities. The cost of a pre-processor system to function, for example on FORTRAN, is an order magnitude less than the direct production of compiler-interpreter systems.

4. Analysis of Components

We turn now to a detailed analysis of the problems and costs as associated with the five component techniques.

4.1 The "state of the art" of rigorous proofs of the correctness of representations of algorithms in programming languages is in an early state. It seems possible within the next two to five year period to bring this to a fairly respectable state of automatic analysis. At the moment, there is much work remaining to be done on identifying the properties of languages and the types of statement structures which simultaneously allow ready application of rigorous mathematical analysis for correctness and ease and convenience of problem formulation. We attach as Appendix A a detailed report on work done in this area under this contract.

4.2 The generation of all possible tasks through modules of modest size can still be a very significant, if not computationally intractable size. It is, however, for most currently used programming languages, a problem which is quite soluble in terms of reasonable costs. It is still, however, important to be able to selectively key upon partial aspects of the flow of control access modules so as to obtain manageable flow graphs for logically coherent concepts. It is also important to be able to key on <u>critical data</u> values which determine flow paths of special interest. A further use of <u>critical data</u> values is to be able, in the flow analysis to specify values, sets of values, or ranges of values, for these variables to further select flow paths of special interest. Appendix B to this report outlines work done under this contract toward the automatic production of flow graphs and the use of critical data values concepts.

4.3 Analysis of data base validity factors into two portions; Validation of explicitly transmitted information and validation of access to global information. The validation of explicitly transmitted information may occur through several mechanisms: by explicit analysis in the source code, by analysis of language defined concepts by the compiler or language interpreter, by hardware analysis of data types and allowed location ranges (transmission of arguments by reference only using a specified range of cells), by embedding of simulation modules or by interpretive execution. This boundary interphase problem is a difficult one in that modular testing isolates the consuming process from the supplying process. This is a problem which has had exceedingly little study from the basic computer science viewpoint. It is clear that a mixture of devices will be needed. The study of access to global data is a special case of the flow path analysis of section 4.2. The flow paths studies can be selected to be those involving sequence of access to common data and the actual access mechanisms. There has been implemented, partially under the support of this contract and partially by the National Science Foundation, a program which carries out analysis of access to data structures by producing a flow graph of the access and sequencing process and comparing it to a flow graph or glow graphs for correct access and sequence mechanisms.

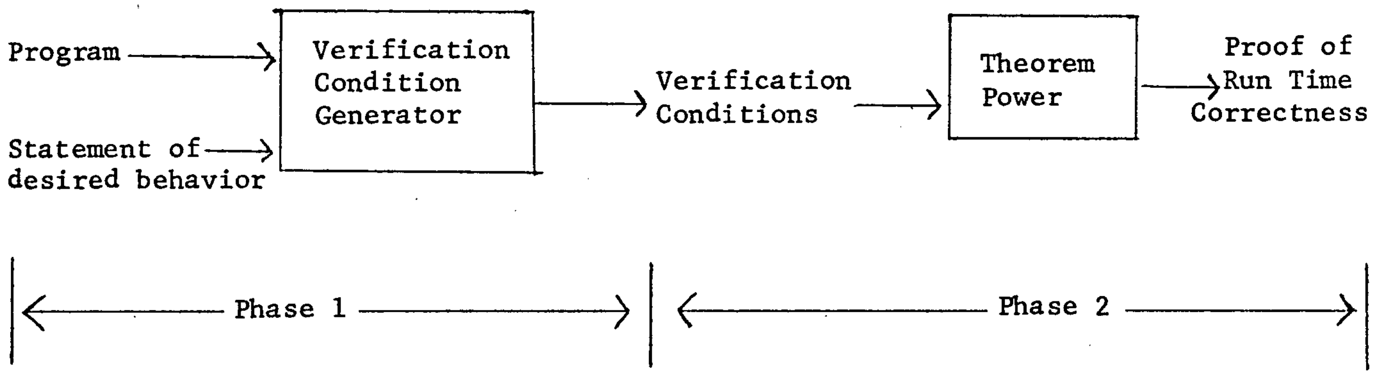
4.4 The use of mixed execution-simulation testing has a fundamental effect on testing strategy. It allows the partial relation of global or "top down" program development. A completely executable program must have all of its modules coded before its key modules can be tested in a complete environment. This usually means that attention is focused on the coding of the individual modules of the design and specification of the environment of the individual modules at the leaves of the tree. The mixed execution-simulation testing allows that modules may be tested by execution at any level on the tree with the balance of the operating environment being simulated. It would typically be the case that development of a simulated module will be as expensive as the development of the actual module. Additionally, the normal circumstance will be that a simulation will have to be carried out in the language used by executable modules. This is a point which needs to be altered. In some cases it will be possible to write a comprehensive environmental simulator without having to reproduce in toto the modular structure of the executable code. It is readily possible to produce an integrated debugging system which would have a run-time compatible problem source and simulation source through the use of a pre-processor as described in Section 3.

4.5 The motivation for the use of an interpreter as a program validation tool is the ease with which traces of system activity can be maintained. The need for an interpreter, however, can be lessened by the use of a pre-processor system which can assist in the generation of partial traces. A particular advantage of an interpreter is that it can be made to interpret the problem code into convenient problem-oriented structures and formats. The construction of an interpreter for rigorously segmented modules of code is far less effort than that of an interpreter for a typical general programming language environment. The use of an interpreter only for small modules of code overcomes the principle problem with the use of interpreters which is their extreme slowness and high cost of operation. The enforcement of an appropriately rigorous modular structure makes the use of interpretation a reasonable tool for program validity testing.

APPENDIX A

A Program to Aid in Proving Program Correctness

Rigorous methods now exist for formally proving that a computer program will behave correctly at run time. Currently, the most generally useful of these is the inductive assertion method which consists of the two phases diagramed below. First, a set of conditions, called verification conditions, sufficient to imply the correctness of the program is constructed by examining the program itself and a statement of how it is to behave at run time. The second step then is to prove that this set of sufficient conditions is satisfied.


Program
Statement of desired behavior
Verification Condition Generator
Verification Conditions
Theorem Power
Proof of Run Time Correctness
Phase 1
Phase 2

Both of these phases can be performed manually for sufficiently simple programs, however, for more complex programs, mechanical assistance is required to cope adequately with the large amount of details involved.

A verification condition generator program for the first phase of this method has been completed. This program is written in SNOBOL4 and operates on programs in a language called Nucleus. This is a simple, but complete, programming language designed in such a way that every program in the language can be rigorously analyzed by the inductive assertion method. The language, which has an ALGOL-like syntax, contains data variables and single subscripted arrays of types integer, boolean and character. All variables are regarded as global variables. The language contains assignment statements, go to statements, if-then and if-then-else statements while statements, two forms of case statements (computed multi-way branches), and procedure calls. Procedures are parameterless, but recursive, and have no local variables. The following example is a Nucleus program for evaluating a polynomial of degree $\leq$ 100 by Horner's method (nested multiplication). The coefficients of the polynomial are contained in the array COEF in order of decreasing powers of x -- COEF[0] is the coefficient of X^{DEGREE} and COEF[DEGREE] is the constant term.

```
INTEGER DEGREE,X;      INTEGER ARRAY COEF[100];   $INPUT VARIABLES$
INTEGER TERM;                                     $INTERMEDIATE VARIABLES$
INTEGER PATX;                                     $OUTPUT VARIABLES$

PROCEDURE HORNER;
PATX := 0;
TERM := 0;
WHILE TERM ≤ DEGREE DO
   PATX := PATX*X + COEF[TERM];
   TERM := TERM + 1;
ELIHW;
EXIT;
START HORNER;
```

The question of stating the correctness of this procedure is handled by making two assertions, one at the beginning and one at the end of the procedure. The first is regarded as an initial assumption and the other as a desired result. The procedure is considered to be correct if the desired result is true whenever the initial assumption is satisfied. The HORNER procedure together with a statement of correctness is shown below. In the assertions, a variable name standing by itself denotes the current value of the variable. A variable name followed immeidately by "0" refers to the value of that variable at the time the procedure was entered. For example, the first assertion in the desired result states that values of X, DEGREE, and the entire array COEF are unchanged by the procedure.

```
INTEGER DEGREE,X;       INTEGER ARRAY COEF[100];  $INPUT VARIABLES$
INTEGER TERM;                                     $INTERMEDIATE VARIABLES$
INTEGER PATX;                                     $OUTPUT VARIABLES$

PROCEDURE HORNER;

ASSERT 0 ≤ DEGREE ≤ 100;

PATX := 0;
TERM := 0;
WHILE TERM ≤ DEGREE DO
   PATX := PATX*X + COEF[TERM];
   TERM := TERM + 1;
ELIHW;

ASSERT X = X.0,  DEGREE = DEGREE.0, COEF = COEF.0;
ASSERT PATX = SUM FROM I=0 TO I=DEGREE OF COEF[I]*X↑(DEGREE-I);

EXIT;
START HORNER;
```

To apply the inductive assertion method, assertions also must be associated with enough intermediate points in the procedure so that all loops are intersected--there must be no path of control in the procedure that can flow from a point p along some path and return to p without first passing by some assertion. In HORNER we can associate assertions as shown below. The assertions express the "partial computation" that the procedure has performed to that point.

```
INTEGER DEGREE,X;      INTEGER ARRAY COEF[100];  $INPUT VARIABLES$
INTEGER TERM;                                    $INTERMEDIATE VARIABLES$
INTEGER PATX;                                    $OUTPUT VARIABLES$

PROCEDURE HORNER;

ASSERT 0 ≤ DEGREE ≤ 100;

PATX := 0;
TERM := 0;

ASSERT X = X.0,  DEGREE = DEGREE.0,  COEF = COEF.0;
ASSERT 0 ≤ TERM ≤ DEGREE+1;
ASSERT PATX = SUM FROM I=0 TO I=TERM-1 OF COEF[I]*X↑(TERM-1-I);

WHILE TERM ≤ DEGREE DO
   PATX := PATX*X + COEF[TERM];
   TERM := TERM + 1;
ELIHW;

ASSERT X = X.0,  DEGREE = DEGREE.0, COEF = COEF.0;
ASSERT PATX = SUM FROM I=0 TO I=DEGREE OF COEF[I]*X↑(DEGREE-I);

EXIT;
START HORNER;
```

Currently, these intermediate assertions must be chosen manually and represent one of the most difficult parts of applying the inductive assertion method. However, the search for these intermediate assertions frequently has beneficial side effects. To find appropriate assertions requires a very thorough understanding of the operation of the program, and attempting to understand the program to the required degree frequently uncovers program errors.

The verification condition generator program produces the following output for the procedure HORNER.

```
INTEGER DEGREE,X;        INTEGER ARRAY COEF[100];  $INPUT VARIABLES$
INTEGER TERM;                                       $INTERMEDIATE VARIABLES$
INTEGER PATX;                                       $OUTPUT VARIABLES$


PROCEDURE HORNER;

(0.1)ASSERT 0 ≤ DEGREE ≤ 100;

(0)PATX := 0;
(1)TERM := 0;

(2.1)ASSERT X = X.0,  DEGREE = DEGREE.0,  COEF = COEF.0;
(2.2)ASSERT 0 ≤ TERM ≤ DEGREE+1;
(2.3)ASSERT PATX = SUM FROM I=0 TO I=TERM-1 OF COEF[I]*X↑(TERM-1-I);

(2)WHILE TERM ≤ DEGREE DO
   (3)PATX := PATX*X + COEF[TERM];
   (4)TERM := TERM + 1;
(5)ELIHW;

(6.1)ASSERT X = X.0,  DEGREE = DEGREE.0, COEF = COEF.0;
(6.2)ASSERT PATX = SUM FROM I=0 TO I=DEGREE OF COEF[I]*X↑(DEGREE-I);

(6)EXIT;
START HORNER



PROCEDURE HORNER   PATH   0 1 2



0.1          0 ≤ DEGREE ≤ 100
-----------------------------------------
0          PATX.1=0
1          TERM.1=0
.........................................
2.1          X = X,   DEGREE = DEGREE,   COEF = COEF

2.2          0 ≤ (TERM.1) ≤ DEGREE+1

2.3          (PATX.1) = SUM FROM I=0 TO I=(TERM.1)-1 OF COEF[I]*X↑((TERM.1)-1-I)



PROCEDURE HORNER   PATH   2 3 4 5 2
2.1          X = X.0,  DEGREE = DEGREE.0,   COEF = COEF.0
2.2          0 ≤ TERM ≤ DEGREE+1
2.3          PATX = SUM FROM I=0 TO I=TERM-1 OF COEF[I]*X↑(TERM-1-I)
-----------------------------------------
2          TERM≤DEGREE
3          0≤TERM≤100        →    PATX.1=PATX*X+COEF[TERM]
4          TERM.1=TERM+1
.........................................
2.1          X = X.0,   DEGREE = DEGREE.0,   COEF = COEF.0

2.2          0 ≤ (TERM.1) ≤ DEGREE+1

2.3          (PATX.1) = SUM FROM I=0 TO I=(TERM.1)-1 OF COEF[I]*X↑((TERM.1)-1-I)
```

```
PROCEDURE HORNER   PATH   2¬ 6
2.1          X = X.0,  DEGREE = DEGREE.0,  COEF = COEF.0
2.2          0 ≤ TERM ≤ DEGREE+1
2.3          PATX = SUM FROM I=0 TO I=TERM-1 OF COEF[I]*X↑(TERM-1-I)
------------------------------------------
2         ¬(TERM≤DEGREE)
....................................
6.1          X = X.0,  DEGREE = DEGREE.0, COEF = COEF.0

6.2          PATX = SUM FROM I=0 TO I=DEGREE OF COEF[I]*X↑(DEGREE-I)
```

A listing of the program is produced in which control point numbers have been inserted. These numbers are used to be able to refer to paths of control through the program and to relate these paths back to the actual program. The second part of the output is the set of verification conditions. Each verification condition has the form

```
Assertions at the beginning
       of the path
. . . . . . . . . . . . . . .
Terms due to traversing the
          path
. . . . . . . . . . . . . . .
Terms due to assertions at the
      end of the path.
```

For a verification condition to be satisfied, the terms above the dotted line must logically imply those below. Also the initial assumption may be applied at any time in proving the terms below the dotted line. For the HORNER example, it can be proved that each of the three verification conditions is satisfied, and hence HORNER is correct.

The verification condition generator that has been completed for Nucleus programs is a first step toward semi-automatic proving of program correctness. The program as it now stands could be applied to a fair number of actual computer programs (provided they were first translated into Nucleus) to assist in making rigorous proofs of correctness feasible. Further automation of the overall method will make proofs feasible for an even larger class of programs.

APPENDIX B

Analyzing Sequences of Operations Performed by Programs

I. Introduction

Many interesting properties of programs can be stated in terms of the sequence of operations they perform. The uninitialized variable problem for example can be described as a violation of the rule: "In any computation of a program using a variable X, some operation which assigns a value to X must precede all operations which use the value of X." A second example is found in communication between processes in a multiprogramming system. Correct communication requires that each process obey a protocol such as that of the "critical section" (1) in which the process must set an interlock before assessing a shared data structure and subsequently must release the interlock. The objective of the research described here is the development and automation of a procedure for verifying that programs obey given ordering rules on the sequences of operations they perform.

Traditional debugging by enumeration of cases breaks down for large and complex programs such as operating systems. The procedure described here uses verification rather than simulation techniques, which is to say that programs are checked out by direct inspection of their source code and are not actually run. It consists of comparing a given source program with a prototype for correct sequences of operations and reporting those parts of the program which cannot be matched with the prototype.

The structure of this paper is as follows. Section II describes the basic idea of the approach, including the description of programs and their computations by state graphs, algorithms for manipulating state graphs, and the overall structure of the verification procedure. Section III describes the realization of the procedure in an experimental analysis program and gives more detail about the algorithms used. A point of special interest about the analysis program is that it has been applied to real programs taken from a real operating system. Section IV summarizes the applications to which the analysis program has been put.

II. Basic Concepts

This section summarizes the theoretical aspects of analyzing sequences of operations. Although a certain amount of formalism is used, proofs and detailed definitions are omitted for the sake of brevity. State graphs and their correspondence to programs are defined first, followed by a general technique called folding for manipulating state graphs. The class of program characteristics analyzable by the state graph technique is defined in terms of folding into a prototype graph. Finally, the overall structure of the analysis procedure is stated and some general considerations are discussed.

A state graph is a directed graph with labeled edges. Several edges with different labels may connect the same pair of nodes, and a node may have several edges with the same label entering or emerging from it. The nodes of the graph correspond to program states, that is, to the distinct combinations of values of all memory cells used by the program. The contents of registers, such as the instruction location or program counter, are also included. The edges of the graph correspond to the sequential transitions of the program from state to state as it runs, and are labeled with the symbolic operations performed.

Formally, a state graph is an ordered triple $G = (S, A, \rightarrow)$ where S is a set (the states), A is a finite set (the operations), and $\rightarrow$ is a subset of the Cartesian product $S \times A \times S$. We write $x \xrightarrow{a} y$ if (x, a, y) is an element of $\rightarrow$.

The state graph of a program explicitly describes the set of computations of the program in terms of paths through the graph. Sequences of operations, or traces, are simply the sequences of edge labels along such paths. Unfortunately, the state graphs of actual programs are quite large if not infinite, so it is impossible to deal directly with them. A general technique called folding is used to compress and modify state graphs in finite space and time.

Folding is the merging together of states while preserving the edges involving them. This is formalized as follows. A homomorphism of a graph $G = (S,A,\rightarrow)$ into a graph $H = (T, A, \rightarrow)$ is a mapping $f\colon S \rightarrow T$ such that

$x \xrightarrow{a} y$ in G implies $f(x) \xrightarrow{a} f(y)$ in H.

If such a homomorphism exists then H is a folding of G. For a thorough discussion of graph homomorphisms, see (2).

It is easily shown that since a folding preserves individual state transitions, it preserves paths and thus sequences of operations. Said another way: If ab...c is a sequence of operations in G, and H is a folding of G, then ab...c is also a sequence of operations in H. Note however that the converse is not true. A folding can introduce spurious traces unless its inverse is also a folding. Whether or not the spurious traces do any harm in the analysis depends on the context in which the folding occurs. The general strategy is to keep the state graphs as folded as possible without introducing undesirable spurious traces.

The foregoing definition gives no hint as to what an appropriate homomorphism is like. In analysis of programs, foldings almost always can be defined by ignoring selected program variables. If the variables of a program are divided into two sets, one to be preserved and one to be ignored, then the program's state set can be considered to be a Cartesian product $S = A \times B$, where A represents the values of the interesting variables and B those of the uninteresting ones. Assuming this has been done, an appropriate folding is the projection mapping f(a,b) = a, which simply discards the uninteresting variables. Actual foldings can be much more selective than this, ignoring variables only in parts of the state graph for example.

Assembly language programs themselves are extreme examples of folding of their state graphs, in which every variable except the program counter is ignored. The states correspond directly to the addresses of instructions (program counter values) and the symbolic instructions label the transitions from state to state. Throughout we label the edges corresponding to conditional branches with the outcome of the branch. The folded state graph directly defined by a program is a simple modification of the program's flowchart, with the operations labeling the edges rather than the nodes.

Folding has several uses in the analysis of programs using state graphs. It allows finite representations of infinite state graphs. It may be used to manipulate a state graph by unfolding it in some ways and refolding it in others to get a more explicit or compact representation of interesting sequences of operations. Last but not least, it provides a characterization of the kinds of properties that can be analyzed using state graphs. The last will be described next, using an example and then a definition.

Dijkstra's "critical sections" are a tool for coordinating asynchronous parallel processes. They are defined in terms of four primitive operations:

P - set an interlock which prevents any other process from performing the P operation until this process has performed a V operation.

V - Release the interlock.

A - Access the shared data structure protected by the P and V operations.

ϵ - Any other (irrelevant) operation.

A process is defined to be correct in its usage of these operations if and only if its state graph can be folded into the following prototype:

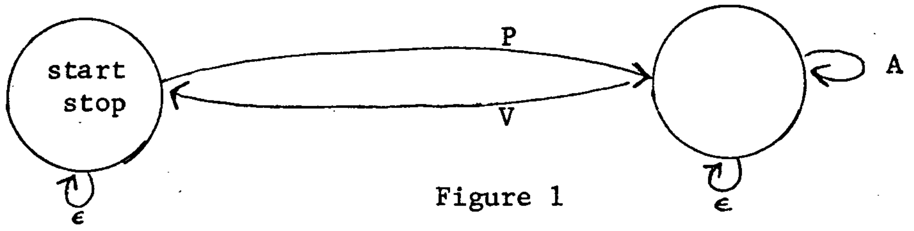


Figure 1

The node at the left represents the noncritical part of the program, and the node at the right is the critical section. The correctness criterion defined by the prototype is that a program may not access the shared data or do a V operation if it is not in its critical section, that it may not halt or do a P operation when it is in its critical section, and that the P and V operations switch the program between the noncritical and the critical sections.

As suggested by this example, the properties which can be verified using state graphs are those which can be stated by defining a prototype into which any correct program should be foldable. All program verification techniques must eventually appeal to intuitively acceptable standards of what makes a program correct. The prototypes play this role here. The objective is not to verify the prototype in any way, but rather to decide if a given program can in fact be folded into the prototype.

The technique for verifying that a program can be folded into a given prototype is divided into three steps:

A) Convert the program's symbolic source deck into the flowchart-like folding which ignores all variables but the program counter.

B) Transform the initial folded graph to represent distinctions in the values of "critical" variables and to eliminate superfluous detail.

C) Attempt to fold the transformed graph into the prototype and report success or failure. If failure is reported, identify the portions of the program which could not be folded into the prototype.

The details of automating these three steps are discussed in section III.

There are, however, several points which deserve further discussion.

Referring back to the critical section prototype, note that all "uninteresting" operations were lumped together in a single operation labeled ϵ . Note also that the prototype allows such operations to be freely interspersed with the interesting operations in any legal sequence. This is a common phenomenon in prototypes and deserves some special treatment as it can be used to fold considerable uninteresting detail out of the state graph.

An operation is "unitary" if it appears in the prototype only on unit loops. (A unit loop is a transition $x \xrightarrow{a} x$ which does not change the state.) It is "uninteresting" if it is unitary and <u>every</u> state of the prototype has such a unit loop. In the critical section prototype, A is a unitary operation and ϵ is uninteresting. An edge of the program state graph is unitary or uninteresting if it is labeled with a unitary or uninteresting operation. Uninteresting edges are also called null edges. Such edges provide a special opportunity for folding the program state graph as follows:

1) Any two nodes connected by a unitary edge may be folded together. The resulting graph will be foldable into the prototype wherever the original was, because the two states in question must fold into the same state of the prototype due to the unitary edge connecting them.

2) Once the above folding has been accomplished, all unitary operations in the program state graph appear as unit loops. For uninteresting operations these loops may be eliminated entirely since it is known that they will be allowed in the prototype.

It should be noted that the general folding technique described above can introduce spurious illegal sequences if an illegal sequence already exists.

For example, consider the effect of adding a null edge in parallel with the V edge in the prototype graph of Figure 1. This corresponds to a program's illegally jumping out of its critical section without releasing the interlock. Theillegal sequences generated contain an excess of P operations. If the two states are folded together then spurious illegal sequences containing an excess of V operations are also generated.

The algorithms described in Section III avoid the generation of spurious sequences by constraining the application of folding. For example, the "only successor" rule allows folding of nodes x and y with the null edge x   y whenever y is the only successor of x. Such restrictions make it considerably easier to reconstruct the illegal path in the source program given an illegal sequence in its folded version.

The need for step B, which unfolds and refolds the state graph is best illustrated by an example. The following fragment of an Algol program contains a disguised critical section.

```
.
.
.
for I: = 1 step 1 until 3 do
begin
if I = 3 then V;
if I = 2 then A;
if I = 1 then P;
end;
.
.
.
```

Figure 2

Here P, V, and A are the primitive operations to be matched against the critical section prototype defined earlier. The initial state graph generated from this program fragment has the form:

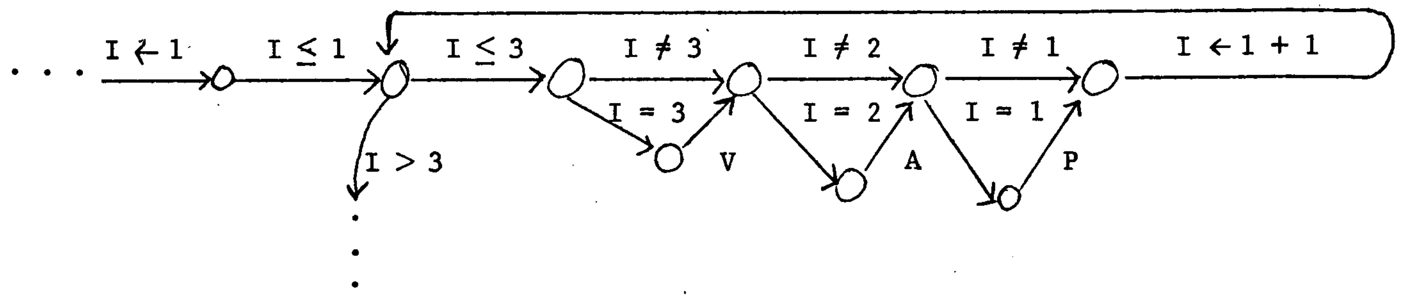


Figure 3

In this example the variable i is critical in the definition of the actual as opposed to apparent flow of control. If the graph were simply folded to eliminate all operations except for P, A, and V the result would be:

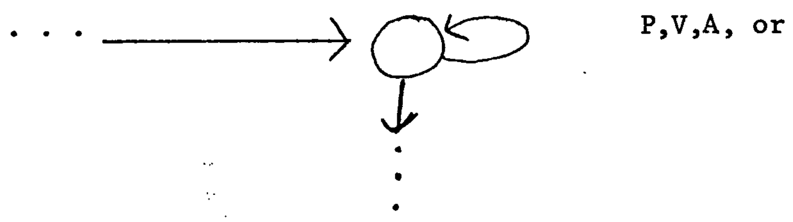


Figure 4

This graph cannot be folded into the prototype. Thus the graph must be unfolded to reflect the distinct values of I, giving the intermediate result:

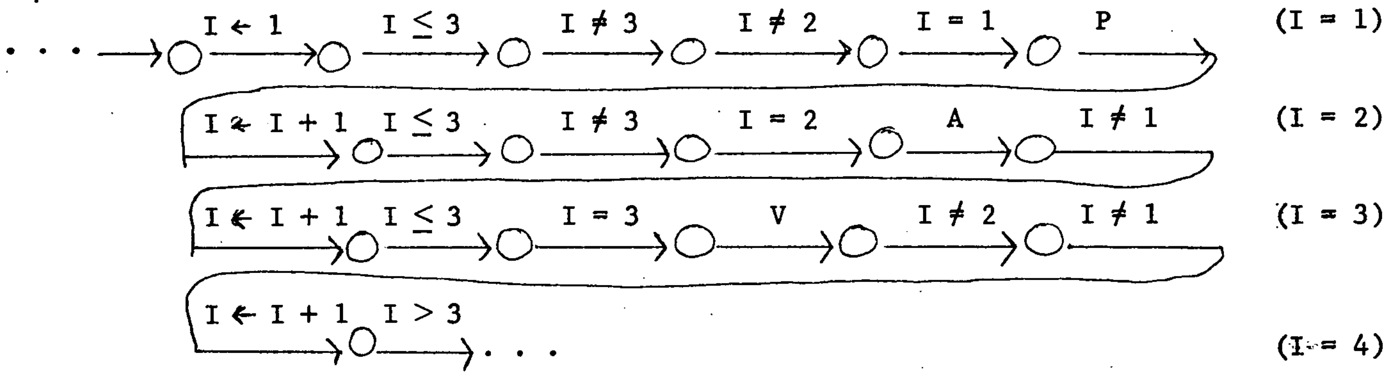


Figure 5

If this second graph is folded by deleting insignificant operations, the result is

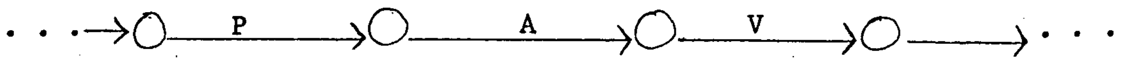


Figure 6

which obviously does fit the prototype.

The point of this example is that the initial folded graph may need to be unfolded for certain variables which play a critical role in determining the sequences of significant operations. The unfolding is done by splitting selected nodes of the graph into separate copies for each different value of the critical variable. This implies that a finite set of such values must be known. This is a restriction on the method, but fortunately critical variables usually satisfy the restriction.

The splitting procedure has been automated with one major omission: the identification of critical variables. This is accomplished manually by input to the analysis program. Overlooked critical variables are quickly found since they result in spurious paths which cannot be folded into the prototype.

One major class of critical variables is identified automatically. This is the class of subroutine returns. State graphs represent subroutine structure as follows. With each subroutine is associated a variable which identifies the points at which the subroutine is called by distinct values. This return variable is set before the subroutine is called, and the subroutine returns to the proper point by branching on the value of the return variable. The return variables are automatically considered to be critical by the analysis procedure, with the result that a separate copy of the subroutine is inserted at each point of call by splitting procedure. The finitude restriction on critical variables implies that recursive subroutine calls cannot be handled.

The final step, folding the transformed state graph into the prototype, is accomplished by a straightforward matching process which defines the desired homomorphism. The process starts by matching the starting state of the program to the starting state of the prototype, and extends the matching to the rest of the graph by propagating matches based on the homomorphism rule that $x \overset{a}{\rightarrow} y$ implies $f(x) \overset{a}{\rightarrow} f(y)$. It is simplified by requiring that the prototype be deterministic. This is easily accomplished manually, either by inspection or by use of the power set techniques borrowed from the theory of finite automata.

III. Techniques

The ideas and methods of the previous section have been implemented in a program called TRACE. TRACE is divided into three independent sections which sequentially perform the major steps in producing a compact state graph for a given object program. BUILD reads the source code and generates an initial state graph. SPLIT resolves questions which arise when the program branches on the values of critical variables. CLEANUP performs final folding and outputs the graph in various forms suitable for external analysis, and may also immediately verify that the object program does or does not conform to the order-of-operation rules specified by the user-supplied prototype graph.

1. Building the State Graph.

The only major inputs to TRACE are the source code of the object program and a list specifying which instructions or operations in the source code are to be considered interesting. This list must always contain all instructions which affect the flow of control within the object program such as jumps, branches, subroutine calls and returns, and program stops and ends. Additional items on the list will depend on the application. In the table interlock example, the operations which reserve, access, and release the table will be included.

In the present version of BUILD if these operations of interest are not explicitly coded in one or two lines or with a macro, the source code must be "doctored" by the user; a unique instruction must be invented to represent the interesting operation, and must be inserted at the appropriate points in the code and included in the list of interesting instructions. Occasionally unobvious code may have to be rewritten; for example, table jumps must be replaced with explicit branches.

BUILD reads the source code one line at a time selecting those lines which have location labels or whose op-code is on the interesting instruction list, and ignores all other lines. For these interesting lines, appropriate additions are made to the state graph. Labels cause the creation of a new node corresponding to the state of the machine just before the labeled instruction is executed. This new node then becomes the "current" node. These program locations labels are attached to the nodes so that later the user may easily see which nodes correspond to which sections of the source code. Jumps cause the creation of a null edge between the current node and the node corresponding to the location jumped to, and the creation of a new current node. Application-dependent interesting actions such as semaphore operations cause the creation of an appropriately labeled edge from the current node to a new node corresponding to the new state of the machine after the interesting action has taken place.

BUILD may be told by the user to keep up with the value of certain critical variables which affect the flow of control within the object program. When these variables are set to a value in the object program, this information is attached to the currect node. When the program branches on the value of these variables, BUILD creates two null edges from the current node, and tags the edges with the condition under which each path is taken. For example, if the variable SWITCH has been designated critical, the segment of source code in figure 7 will result in the graph segment of figure 8.

```
               .
               .
          JPOS       LOC1          branch to LOC1 if (acc) ≥ 0
          P          J             reserve table J
          LOAD       1
          STORE      SWITCH        SWITCH   1
          JUMP       LOC2          unconditional jump to LOC2
               .
               .                  (uninteresting code)
               .
LOC1      LOAD       0
          STORE      SWITCH       SWITCH   0

               .
               .
LOC2
               .
               .
          LOAD       SWITCH
          JZERO      LOC3         branch to LOC3 if SWITCH = 0
          V          J            release table J
LOC3           .
               .
               .
```

Figure 7

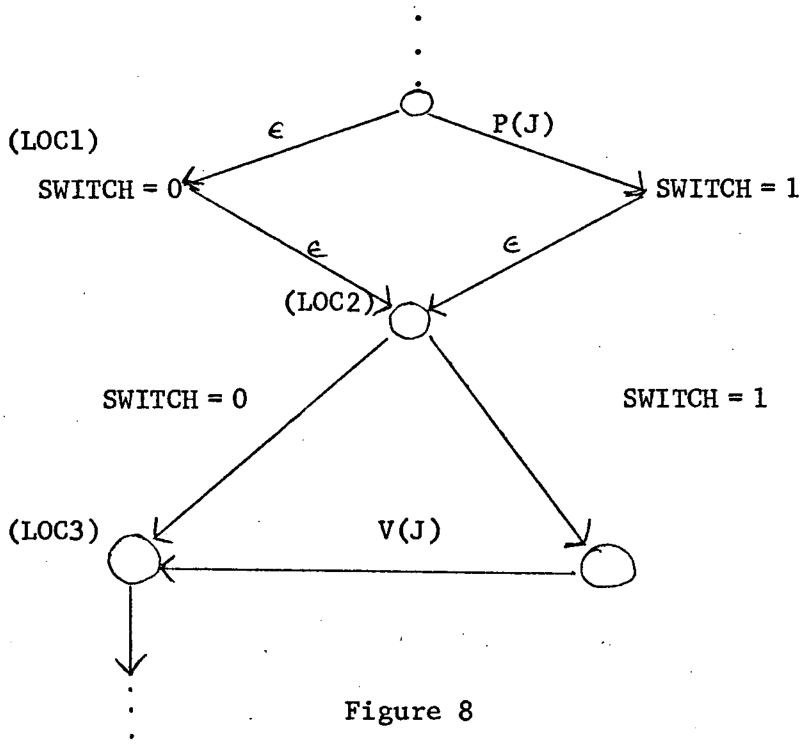


Figure 8

When a branch is made on a value which BUILD has not been told is critical, two untagged edges result, and the state graph contains a non-deterministic branch such as the one from the top node of figure 8. BUILD automatically handles subroutine calls and returns as setting and branching on, respectively, the value of a generated critical variable.

2. Splitting.

The directed graph produced by BUILD contains complete information about the sequence of operations which the ojbect program does, or may, perform, but it will generally be too large and too complex to be useful. The folding techniques for reducing the size of the graph are described in III.3. The complexity arises because of the conditional edges which result from subroutine returns and from branches on other critical variables. SPLIT resolves such conditional paths by creating two or more copies of the graph segment corresponding to the (possibly) different sequences of actions performed as the critical variable takes on its range of values. This splitting process begins at the node from which there are conditional edges and propagates backward along the directed graph to the nodes at which the critical variable acquired known values. For example, from the graph segment of figure 8, SPLIT will produce the graph segment of figure 9.

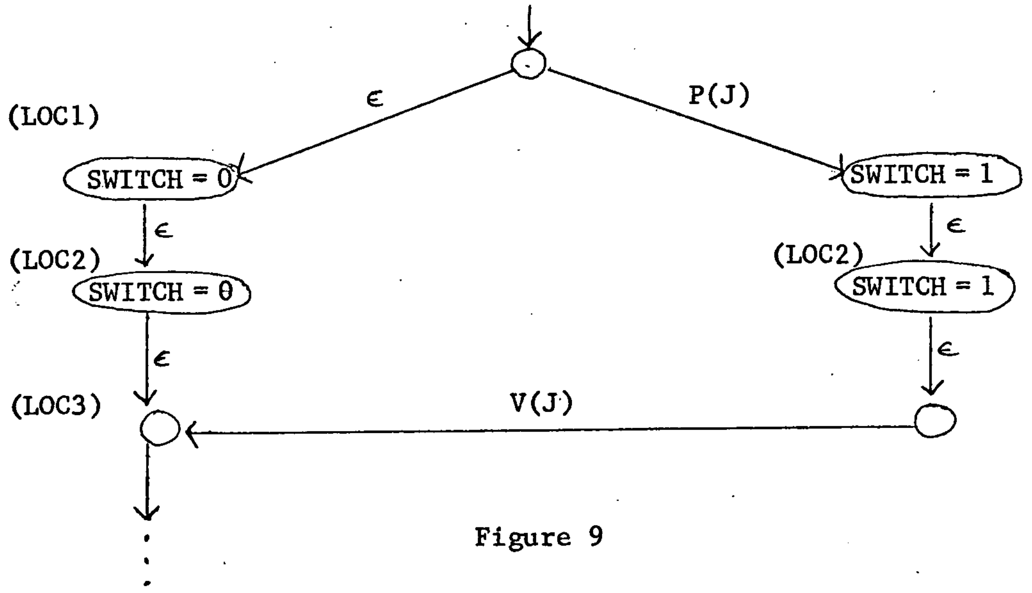


Figure 9

SPLIT keeps a pushdown stack of "question nodes", nodes from which there are conditional edges. This stack is initialized to contain all the question nodes in the graph originally produced by BUILD. SPLIT begins each execution of the splitting algorithm by considering the top node of the stack. If it does not have associated with it answers to all of the questions on its outgoing edges, this node is split, the questions propagated back onto its incoming edges, and all of its predecessor nodes are pushed onto the stack. Loops are handled by marking each copy of the split node with the corresponding value of the critical variable. When the stack is empty, SPLIT terminates.

3. Folding.

The initial state graphs produced by BUILD can be quite large; for example the graphs representing operating system programs written in CDC6600 periphial processor assembly language average about one node for every three lines of executable source code, and about 1.5 edges per node. (These ratios will probably vary depending on the source language and nature of the program.) The splitting algorithm greatly increases the number of nodes. Thus both to conserve memory and in order to reduce the amount of work to be done by subsequent portions of TRACE, it is imperative to reduce the size of the graph as much as possible at each stage of the process. This reduction is accomplished by folding. Folding may introduce new traces into a graph, and although the general folding rule given in Section II will never introduce spurious illegal traces into a graph which did not already contain at least one, the folding rule can be modified so as never to introduce them. It can be easily shown that applying the following two rules will never introduce spurious illegal traces: i) If node I is directly connected to node J only by a null ( ) edge from I to J, and either a) I has no other

successors, or b) J has no other predecessors, then I and J may be combined. ii) If there is a null edge from node I to J and a null edge from J to I, then I and J may be combined.
In addition, null edges from a node to itself, and one of a pair of identical edges between the same two nodes, may be eliminated. When two nodes are combined, location labels and information regarding the value of variables attached to either of them are attached to the combined node, with duplications eliminated. Nodes connected by a conditional edge are never combined.

The above rules are incorporated into a routine called FOLD which applies them in turn to each node in the graph. If any combinations were made in a pass through the whole graph, then FOLD again applies the rules to every node, and the process is repeated until a pass has been made in which no new combinations occurred. Applying FOLD to the graph segment of figure 9 produces the graph segment of figure 10.

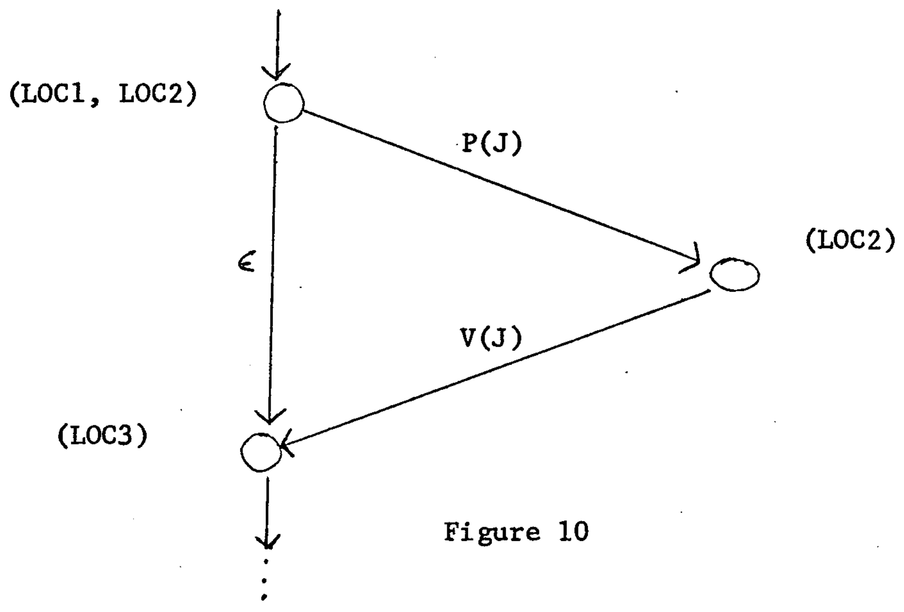


Figure 10

FOLD is applied to the graph as soon as BUILD has finished and is usually able to cut its size in half. During the execution of SPLIT,FOLD may be called whenever memory gets crowded, and is always called when SPLIT has finished. It has been found that at this point even very large object programs will be represented by graphs having no more than 40 nodes, and usually less. It therefore becomes practical in CLEANUP to apply folding rules with more sophisticated criteria for combining states:

iii) Any two nodes whose sets of outgoing edges are identical, with respect to both labels and successor nodes, may be combined.

iv) If there is a closed path, no matter how long, consisting entirely of null edges from a node back to itself, then all the nodes along this path may be combined into one node. (Notice that this is a generalization of rule ii; in rule ii the length of the path is just two)

When all of the folding rules have been applied to every node in the graph without any new combinations occurring, the graph is completely folded. CLEANUP will then output the final graph and may also try to match the graph against a prototype graph.

IV. Applications and Conclusions

The program called TRACE, described in the previous section, is an experimental version of what we hope will become a useful tool for program verification and system debugging. Its primary purpose so far has been to verify that the techniques described in this paper are in fact practical. In its present form TRACE is written in FORTRAN and requires 4200_8 words of memory on a CDC 6600. From the runs that have been made, it appears that TRACE requires less than 2 seconds of central processor time per 100 lines of source code to produce the final state graph for a program.

Despite its experimental nature, TRACE has been used to verify one aspect of the UT2 operating system used at the University of Texas Computation Center. TRACE has been run on 6 of the system programs which access the Job Status Table to verify that they all adhere to the semaphore protocol. It can be used in its present form to verify that all programs adhere to the correct protocols for reserving channels, disk and ECS space, etc.

The state graphs produced by TRACE are of use in connection with a major performance measurement and evaluation project currently in progress on the UT2 operating system. This project includes an event driven trace which eventually produces directed graphs representing sequences of actions actually taken by programs including system programs (3). All these events can be detected by TRACE in the source code of the same system programs. Thus the graphs produced by TRACE can be used to preset the event trace. In addition, comparison of the two graphs produced by these two different methods helps to verify each method and may indicate sections of source code seldom or never executed, sections which are heavily used, etc.

Most of the problems encountered so far in implementing the techniques described in this paper arise in the input and recognition phase in which the initial state graph is produced from the source code. Not all interesting actions are automatically recognized. The value of critical variables may be set by an input statement. At present such problems must be handled by modifying the source code. An average of about one modification per 75 lines of source code have had to be made by hand to enable TRACE to correctly recognize all interesting operations.

Another area for future research is language independence. The fact that source code operations which are to be considered interesting are input at run time indicates that TRACE could be run on programs written in different source languages with little or no modification, but this hypothesis has yet to be tested.

A great deal of research needs to be done to determine just how many interesting properties of programs can be stated purely in terms of the sequences of operations they perform, and for those properties which cannot be so described, whether or not any state graph techniques can be applied to them.

REFERENCES

1. Dijkstra, E. W. Cooperating Sequential Processes, In Programming Languages (F. Genuys, ed.) Academic Press (1968), 43-112.

2. Hedetniemi, S. T. Homomorphisms of Graphs and Automata. Technical report, Communication Sciences Program, The University of Michigan, 1966.

3. Sherman, S., Howard, J. H., and Browne, J. C. A Comparison of Deadlock Prevention Schemes using a Trace-Driven Model. Sixth Annual Princeton Conference on Information Sciences and Systems, Princeton, N. Y., March 1972.

C.2

III. Redundancy Techniques: Roll-Back and Recovery

Introduction

Rapid and smooth restoration of a computing system after an error or malfunction is always a major design and operational goal. Hardware failures can be detected and corrected by suitable diagnostic and maintainability procedures. Software design errors could be hard to detect but once detected corrective procedures would be easy to implement. The majority of operational failures occur in the hardware processors, memory, and I-0. On-line diagnosis and use of error checking codes, have been effective in reducing the effects of these hardware malfunctions. After the malfunction is corrected, the problem arises as to where to restart the program. It may not always be feasible to run the entire set of programs again from the start, either due to time limitations or since the required data has been modified. A better strategy would be to have a number of roll back points (or check points) within the program at which certain program and processor status information could be saved. If a fault or malfunction is detected, the program is rolled back to a previous check point where the system is known or proven to be in good operational condition.

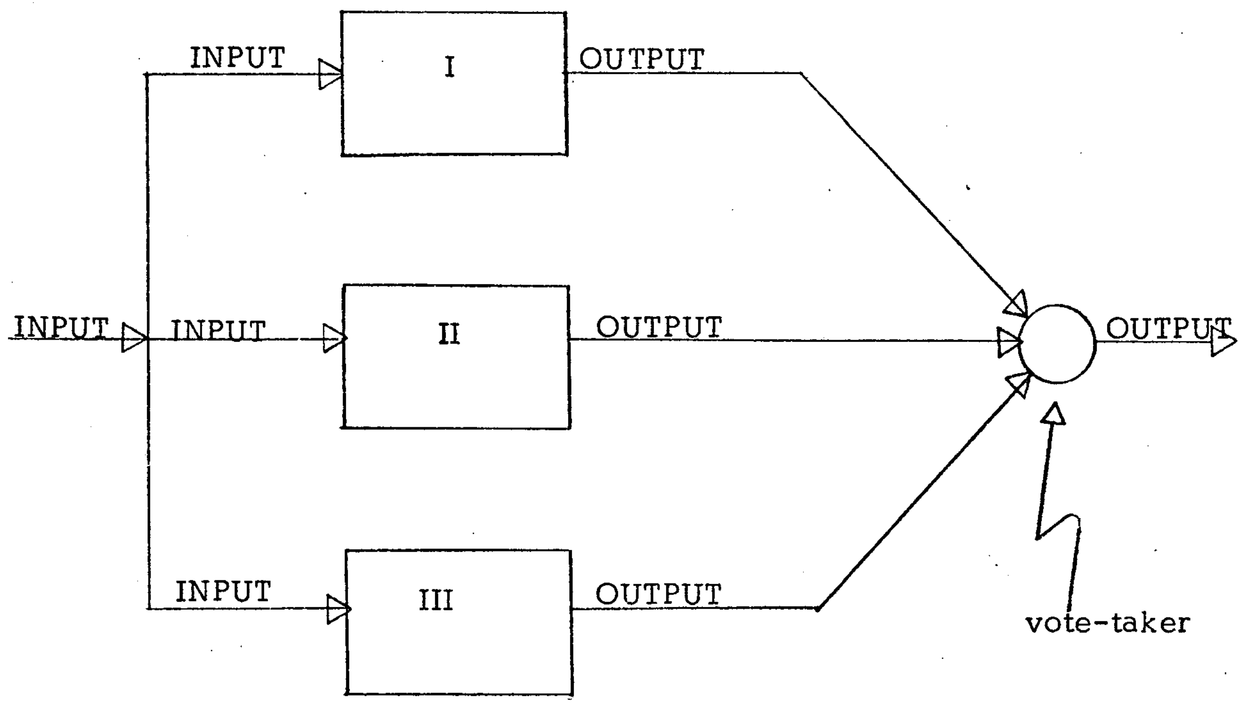
Various strategies are used to reduce the impact of interruptions or malfunctions both to the system and to the users. Operating System 360 as used in Model 65 is equipped with a set of programs called the Recovery Management Support which embodies a number of methods. The recovery methods depend upon the nature of the malfunction. In the I/0 area, re-reading of input data with parity errors is common. If error subsists even after repeated retries the system could consider reconstruction of damaged data (error correction) if possible. In the case of the processor errors, the instruction may be retried if feasible (i.e., if its operands were not modified by the instruction). The most important technique is to provide check points in all programs so that programs could be rolled back to a previous state and computation resumed.

If an error is detected while a program is being processed and if the error cannot be corrected immediately, it may be necessary to run the entire program again. The time lost in running the program again may be substantial and in some real time applications (notably aerospace and process control) critical.

At any stage in the processing of a program certain information is required by the program for computation to proceed successfully. A state at any stage in the processing of a program, will be defined as the information (variables, data, programs..) which may be subsequently used by the program. Saving the state of a program is the process of making a copy of the state in secondary storage. Clearly, the length of time spent in saving a state is proportional to the amount of information that has to be copied. Loading a saved state is the process of setting all the registers, primary and secondary storage etc. to the values stored in them when the state was saved. Recovery time can be reduced by saving states of the program at intervals, as the program gets processed; if an error is detected the program is restarted from its most recently saved state. If the states of the program are saved too frequently, an unnecessarily large amount of time may be spent in saving states. If the states of the program are saved too infrequently an unacceptably large recovery time may result. The resolution of the tradeoff is the subject of our discussion.

Only transient malfunctions are treated in this paper. It is clear that permanent malfunctions cannot be treated by rollback alone, since, if a permanent malfunction does occur and is detected, and the system starts recomputing from a rollback point, the very same permanent malfunction will be detected again. On the other hand, rollback is a very useful tool for handling permanent malfunctions, when it is used with some other fault-tolerant technique which effectively switches off the malfunctioning device. This is discussed in greater detail below.

We will now discuss some areas where rollback can be profitably combined with other fault-tolerant techniques. The earliest attempts at obtaining ultra-reliable systems attempted to achieve reliability through redundancy(4). Triple Modulo Redundancy (TMR) and other methods of fault-tolerant computing using several identical computing units, operating in parallel on the same data, with a vote taker, (see figure 1) have been discussed in great detail(5,6,7). A slightly different system, using TMR and stand-by spares which are switched in when needed, has been described by Mathur and Avizeinis(8). This system



Triple-Modulo Redundancy

Figure 1

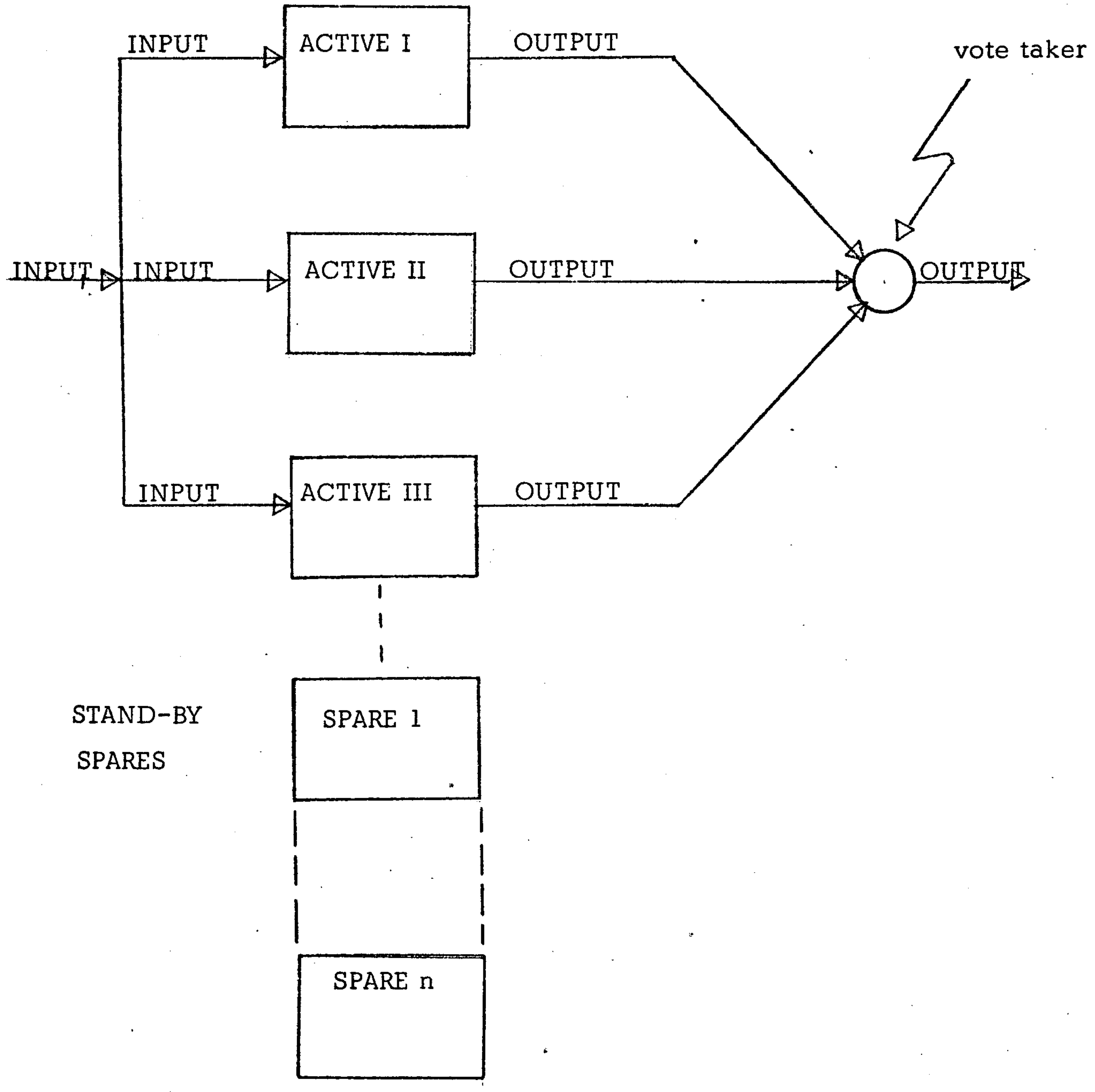
is called the hybrid system. The operation of the hybrid system, in brief, is as follows: three identical computing units are operated in parallel, and a vote-taker compares the outputs of each unit with the others (see figure 2). If the output of one unit does not tally with the other two, it is switched out and a stand-by spare is powered on to take its place. However, after the standby unit is powered on, the registers, memory, and program status word must be loaded with the appropriate information, before processing can continue. One way of doing this is to have rollback points; the three units, including the stand-by unit are loaded with the information saved at the last rollback point, and processing continues from there.

Any system which uses spares is confronted with the problem of loading the powered-on spare. One method for solving this problem is to use rollback. Rollback at periodic intervals was used in the SABRE 7090 System, and in the IBM 9020 System used by the FAA.

We will make the assumption that if an error occurs while a task is being processed, then the error is diagnosed before the task is completed. Suppose an error occurs while a task is being processed and suppose it is not diagnosed; if there is a rollback point immediately after the task is complete, then the information which is saved at the rollback point will be faulty. Subsequently, if the error is diagnosed, this <u>faulty information will be loaded,</u> and the computer will continue processing from the rollback point; eventually the same error will be diagnosed again.

If the same error is detected after rolling back, the system should conclude that there is either a permanent malfunction, or that an undiagnosed error occurred before the last rollback point. The program may be rerun from the very beginning and if the same error is detected again, one may reasonably conclude that a permanent malfunction has occurred, and a reconfiguration made to switch off the faulty unit.

Rollback can be used in two quite different ways. In some systems, the programmer preanalyzes his program and specifies where rollback points are to be inserted. He may decide where to insert rollback points in either an intuitive manner or by making estimates about relevant parameters in his program (such as the maximum time that may be required to process a given task in the



Hybrid System.

Figure 2

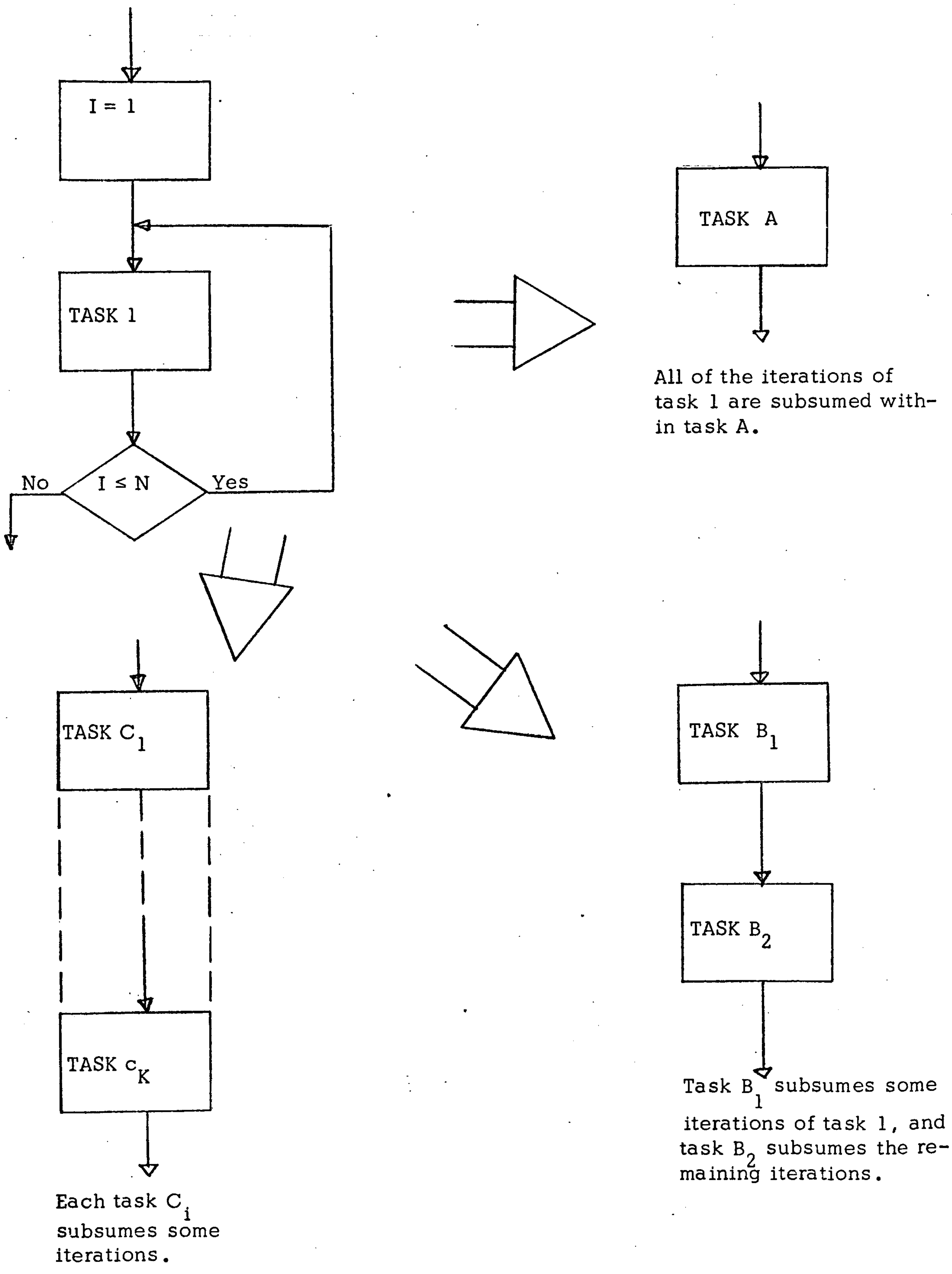
program), and by using a mathematical model to aid him in the decision making. In other systems rollback points are inserted at periodic intervals[14], irrespective of the particular programs being run. We are concerned with the former case, where rollback points are tailor-made for a particular program.

The amount of information that has to be saved so as to be able to restart a program at that point may vary widely from one point in the program to the next. We assume that there is a sufficient amount of secondary storage to store the state of a program at any time. The secondary storage used may be a large core storage unit[9], drum, disk, or even magnetic tape. The "cost" associated with a rollback point is the time taken to save the state of the system at that point; the time clearly depends on the amount of information that has to be stored and on the type of memory used to store the saved state. These factors are included in the mathematical model described later.

This paper uses a graph model to describe a program. Graph models have been dealt with extensively in the literature, see 10, 11, 12, 13. Programmers have traditionally used flow charts (which are graphs of a kind) as aids in programming. In this paper, we assume that a programmer can analyze his program (or flow chart), and represent it as a sequence of tasks. A task may be an instruction, or several instructions including conditional branches. In our paper, we will generally make a coarse partition of the program into tasks, i.e., each task will consist of several instructions and will involve a substantial amount of processing time; the range implied by "substantial" depends on the model used and will be discussed later.

The sequence of tasks processed may change from one run of the program to the next due to conditional branch statements. However, we shall assume that no task is repeated; if a task is iterated in a program, each iteration of the task may be considered a distinct task, or the iterations may be coalesced in the manner shown below. Consider the flowchart in figure 3, where a task is iterated n times. The iterations may be coaelesced into one task, or into a sequence of one or more tasks; each of the tasks in the program graph may correspond to several iterations of the task in the flowchart.

Figure 3



All of the iterations of task 1 are subsumed within task A.

Each task C_i subsumes some iterations.

Task B_1 subsumes some iterations of task 1, and task B_2 subsumes the remaining iterations.

The algorithm makes use of estimates made by the programmer, on the maximum amount of processing time required by a task. Admittedly, it is impossible to design an algorithm, which, given any program, determines the maximum time that may be required to process each task in the program. However, it is possible for a programmer, to obtain estimates of worst case bounds for his particular program. Indeed, in many computer installations, programmers have to submit estimates of the maximum time required to process their jobs. It is important to note that in installations where a programmer is allowed to specify rollback points, he must make estimates of this sort, and then make intuitive decisions based on these estimates. Our objective is to clarify, quantify and formalize his decision-making process. The accuracy of the decisions (intuitive or formalized) clearly depends on the accuracy of the estimates.

Obtaining a program graph from a program is not inexpensive. The programmer must analyze his flowcharts and make estimates of several parameters. Many (probably most) programs are short enough so that no rollback points at all are required. In many other cases, the advantage gained in having tailor-made rollback points is not worth the time spent by a programmer in obtaining a program graph; in these cases rollback points at fixed intervals are sufficient. However, there are some cases, where the costs of slow error recovery are high, where the system runs a comparatively small set of programs over and over again, and where the advantage of tailor-made rollbacks outweighs the time spent by the programmer in constructing the program graph. We are concerned with cases of the latter type.

The decision to insert rollback points clearly depends on the importance of speedy error recovery, i.e. the penalty incurred if a program does not run to completion in a prescribed amount of time. In some real time applications, it is critical that a program run to completion in some given amount of time, whereas, in most commercial applications the loss incurred if an error occurs is just the computer time wasted.

A programmer has to analyze his flow chart and represent it as a sequence of tasks (program graph) only once. Hence, the greater the number of times a program is run, the more the benefit of tailor-made roll-back points. So, the decision to have tailor-made rollbacks clearly depends on the expected number of times the program will be run.

Programs with short processing times do not need rollback points at all. Thus, a program that is worth analyzing for tailor-made rollbacks must have three characteristics:

(1) The program must require a substantial amount of processing time.

(2) The application of the program must be such that quick error recovery is crucial.

(3) The same program must be run a large number of times.

Problem Formulation

A program will be represented by an undirected graph where vertex i corresponds to task i and edge (i,j) exists if and only if task i is followed by task j with non-zero probability.

Associated with vertex i of the graph is a real number t_i which is the maximum (or expected) time between the start and the completion of task i.

Associated with each edge (i,j) of the graph are two real numbers: S_{ij} and L_{ij}. The state of the program soon after task i is completed and before task j is started (if task j is processed next) is described by the program status word, register contents, primary and secondary storage contents and so forth. The time taken to save (make a copy of) the state of the system at this stage in the program in secondary storage, is S_{ij}. We shall refer to S_{ij} as the save time. The time taken to load the state of the system from secondary storage to primary storage is L_{ij}.

At each edge (i,j) we may choose to insert, or to not insert a roll-back point. If a roll-back point is inserted at edge (i,j) then after task i is completed, and if task j is to be processed next, the state of the system is saved in secondary storage before task j is started and any prior state which has been saved earlier is erased. Subsequently, if a transient error occurs, the program is restarted at the most recently saved state.

We define the recovery time r at any point P in the program to be the time taken to load the most recently saved state, and to rerun the program from this state to P. If an error is detected at point P, the recovery time r is the time "lost" due to the error. The question that we wish to answer is: Where should rollback points be inserted?

There are several formulations of the problem. Three of the models are discussed below. In all models we assume that if an error occurs while task i is being processed then the error is detected before task i is completed.

Worst Case Design

Data: With every task i we associate a real number t_i, where t_i is the maximum processing time that will be required by task i. L_{ij} and S_{ij} are the maximum load and save times if a rollback point is inserted on edge (i,j). We are also given M, the maximum recovery time.
Constraints: Insert rollback points so that at every point in the program the maximum possible recovery time does not exceed M.
Objective Function: Minimize the maximum time (i.e. for the worst case) that may be spent in saving states of the system in secondary storage.

Minimal Expected Save-Time Design
Data: Associated with task i is a real number t_i where t_i is the expected time required to process task i. A real number p_{ij} is associated with each edge (i,j) of the graph, where p_{ij} is the probability that task i will be immediately followed by task j. L_{ij} and S_{ij} are the expected load and save times if a rollback point is placed on (i,j). We are also given M, the maximum expected recovery time.
Constraints: The expected recovery time at any point in the program is not to exceed M.
Objective Function: Minimize the expected time spent in saving states of the system in secondary storage.

Minimal Expected Run Time Design

Data: p_{ij} is a real number associated with each edge (i,j) where p_{ij} is the probability that task i is immediately followed by task j. We associate a probability Q_i with task i where Q_i is the probability that a transient error will occur while task i is being processed. Given that a transient error does occur while task i is being processed, let y_i be the time between the initiation of task i and the occurrence of the error. y_i is a random variable; we assume that the probability distribution function for y_i is known. Given that a transient error will not occur while task i is being processed, let t_i be the time required to process task i. t_i is a random variable; we assume that the probability distribution function for t_i is known. We shall assume that the save and load times, S_{ij} and L_{ij} are constants.

All events are assumed to be independent.

Constraints: None

Objective Function: Minimize the expected run time of the program.

A Comparison of the Different Formulations

A programmer can generally provide an estimate of the maximum time that a task will require to get processed; he usually finds it more difficult to estimate the probability distribution function for the processing time required by any given task. For this reason, it is not possible to use the minimal expected run time design unless a substantial amount of measurement can be carried out on the program so as to estimate the distribution functions for processing times of all the tasks.

The estimation of the probability that the program will branch in any particular direction is also difficult, without substantial measurement. For these reasons the worst case design is the most pragmatic method of designing rollback points when there are few statistics available.

The best model to use depends on the function of the program as well as on the information available.

The worst case design and the constrained expected recovery time design are discussed in this paper. The minimal expected run time design is the topic of a subsequent paper.

We shall first consider worst case design.

Implementation

It is not possible to predict precisely how much processing time a given task will require. It therefore seems desirable to make insertion of rollback points a dynamic procedure; on some runs of a program it may be preferable to have a rollback point on a particular edge while on other runs of the same program (with different data) it may be preferable not to have a rollback point on that edge. However, the procedure for making the decision on inserting rollback points should be simple so that the decision can be made in real time with little overhead. The method suggested here fulfills these requirements.

We interrogate the recovery time (r) after each task completion and use it as a basis for making the decision on placing rollback points. r can be determined readily: Let D be the clocktime at the end of the last rollback, L the time required to load the system at the last rollback point, E=D-L, and "clock" the current clock time. Then

$$r = \text{clock} - E.$$

Suppose that at some point in the program the task just completed and the task to be processed next are i and j respectively. Let r be the recovery time at this point. We show that the optimal decision is to insert a rollback point if $r > B_{ij}$ and not to insert a rollback point if $r \leq B_{ij}$, _where_ B_{ij} is a _constant_. The set of B_{ij} are computed before the program is run. When task i is completed and if task j is to be processed next, r is compared with B_{ij} and a rollback point is insertèd if $r > B_{ij}$. If a rollback point is inserted, then E is updated. Task j is then processed. A block diagram is presented in Figure 4.

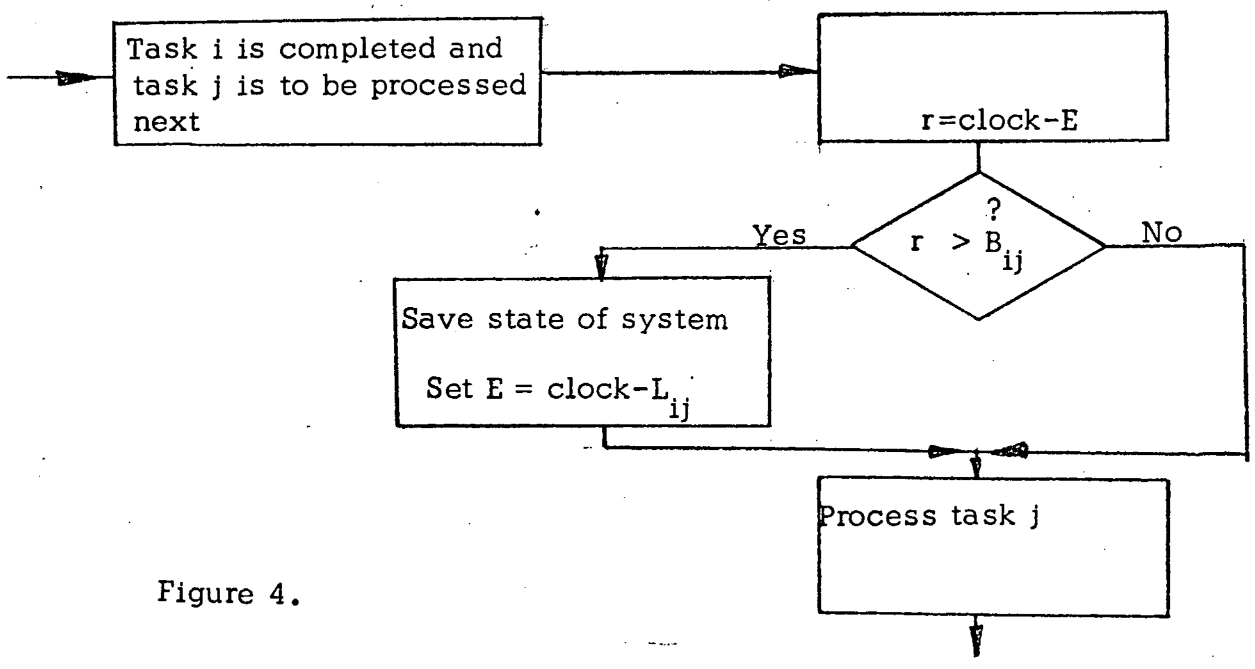


Figure 4.

In general r will vary from one run of the program to the next, since the time required to execute a task will depend on the input parameters. Hence the insertion of rollback points will also vary from run to run of the program, since the decision to insert rollback points is based on the value of r.

Definitions

If there exists a path from vertex i to vertex j then vertex j is said to be a successor of vertex i. A vertex with no successors is called an exit vertex.

For each vertex i in the graph we determine a function $f_i(r)$, for all possible values of recovery time r, where $f_i(r)$ is the minimum time spent in saving states of the system after task i is completed and before the completion of the program, in the worst case. Since we are using worst case design, when predicting the amount of time required by a task i we always assume the worst, i.e., that task i will require the maximum processing time t_i. Similarly, in predicting the branch that a program will take, we assume the worst, i.e. that a program will branch such that the largest amount of time will be spent in saving states of the system.

For each edge (i,j) in the graph we determine functions $g_{ij}(r)$ and $x_{ij}(r)$: $g_{ij}(r)$ is the minimum time spent in saving states of the system after task i is completed and before the completion of the program, in the worst possible case, if task i is followed by task j.

$x_{ij}(r)$ is the optimal decision variable; $x_{ij}(r) = 1$ if a rollback point is to be inserted on edge (i,j) and $x_{ij}(r)=0$ otherwise.

The Algorithm We assume $f_i(r) = \infty$ for all i, and $r > M$.

Initialization (0th step) Define $f_i(r) = 0$, $r \le M$, if i is an exit vertex. Label all exit vertices. (Vertex i is labelled to show that the function $f_i(r)$ has been determined for it)

kth step, k = 1,2,3,....

Determine if there exists any vertex which has all of its successors labelled. If no such vertex exists, STOP, the algorithm terminates. If such a vertex exists let it be vertex i.

For all edges (i,j) compute $g_{ij}(r)$ and $x_{ij}(r)$ from

$$g_{ij}(r) = S_{ij} + f_j(L_{ij} + t_j) \text{ if } r + t_j > M$$
$$= \min \{ f_j(r + t_j), S_{ij} + f_j(L_{ij} + t_j) \} \qquad (1)$$
$$\text{if } r + t_j \leq M$$
$$x_{ij}(r) = 0 \text{ if } g_{ij}(r) = f_j(r+t_j) \qquad (2)$$

= 1 otherwise; B_{ij} is the value of r below which $x_{ij}(r) = 0$.
It follows then that $x_{ij}(r) = 0$ for $r \leq B_{ij}$ and $x_{ij}(r) = 1$ for $r > B_{ij}$.
Then compute $f_i(r)$ from

$$f_i(r) = \text{maximum over all edges } (i,j) \text{ of } \{g_{ij}(r)\} \text{ for } 0 \leq r \leq M \qquad (3)$$

Label vertex i to show that $f_i(r)$ has been computed.
If $f_i(r) = \infty$ for all $0 < r$, STOP. There does not exist any feasible solution to the problem. Otherwise, go to the k + 1th step.

We shall show that the algorithm terminates when all the nodes in the graph have been labelled. We may assume without loss of generality that there is only one entry vertex to the program graph, i.e. there is only one vertex which has no predecessors. Let the entry vertex be vertex number 1; then task 1 is executed immediately after the program is loaded. Let the time taken to load the program initially be L_o. Then, the maximum recovery time immediately after task 1 is completed is $L_o + t_1$ and hence the worst case cost (time spent in saving system environments) will be $f_1(L_o + t_1)$. If $f_1(L_o + t_1) = \infty$ then there is no feasible solution to the problem.

We claim that the optimal decision rule for inserting rollback points on edge (i,j) is given by $x_{ij}(r)$, i.e., if the recovery time immediately after task i is finished is r, and if task j is to be executed next, save the system environment if and only if $x_{ij}(r) = 1$.

Lemma 1: The algorithm terminates when all the vertices in the graph have been labelled.

Proof : Every subgraph of a circuitless graph is also circuitless. A circuitless graph has at least one vertex with no successors; this is true in particular for every subgraph of unlabelled vertices, i.e., in every subgraph of unlabelled vertices, there is at least one vertex which does not have a successor which is also an unlabelled vertex. All the vertices without any successors at all are labelled on the 0th step. Hence, if there exists one or more unlabelled vertices on the Kth step of the algorithm, K=1,2,.. then there must be an unlabelled vertex, all of whose successors are labelled, in which case the algorithm will not terminate. Hence the algorithm terminates only when all the vertices are labelled.

Corollary The algorithm terminates within n steps where n is the number of vertices in the graph.

Theorem 1. The algorithm determines the optimal decision rules $x_{ij}(r)$ for each edge (i,j).

Proof: We shall show by induction on the kth step that if vertex i is labelled on the kth step, then $x_{ij}(r)$, $g_{ij}(r)$ computed by eqs. (1), (2), (3) satisfy the definitions given earlier.

Basis: If vertex i is labelled on the first step, then equations (1), (2), and (3) reduce to

$$\begin{aligned} x_{ij}(r) &= 0 \text{ if } r + t_j \le M \\ &= 1 \text{ if } r + t_j > M \\ f_i(r) \equiv g_{ij}(r) &= 0 \text{ if } r + t_j \le M \\ &= S_{ij} \text{ if } r + t_j > M \end{aligned}$$

A rollback point must be inserted on edge (i,j) if $r + t_j > M$, for otherwise the recovery time after task j is completed may exceed M. A rollback point need not be inserted on edge (i,j) if $r + t_j \le M$, and if vertex j is an exit vertex. Hence, the theorem is trivially true for k = 1.

Induction Step Assume the induction hypothesis to be true for k = 1, 2, ..., t-1. We shall prove it to be true for k = t.

If a rollback point is inserted on edge (i,j) the maximum recovery time after task j is $L_{ij} + t_j$. If the recovery time after task j is completed is $L_{ij} + t_j$, then the minimum time spent in saving states of the system after task j, in the worst possible case, is $f_j(L_{ij} + t_j)$ by the induction hypothesis. S_{ij} units of time are spent in saving the state of the system between tasks i and j. Hence the minimum time spent in saving states of the system, after task i is completed and ii task j is processed next, in the worst case is

$$S_{ij} + f_j\ (L_{ij} + t_j)$$

If a rollback point is not inserted on edge (i,j) the maximum recovery time immediately after task j is $r + t_j$. Hence, in this case the minimum time spent in saving states of the system after task i, and if task j is processed next, in the worse case, is

$$f_j\ (r + t_j)$$

If $r + t_j > M$, a rollback point must be inserted on edge (i,j) if the recovery time after task j is completed is not to exceed M.

If $r + t_j \leq M$, we have the option of not inserting a rollback point ($x_{ij}(r) = 0$) in which case the minimum time spent in saving states of the system in the worst case is $f_j\ (r+t_j)$, or of inserting a rollback point ($x_{ij}(r)=1$) in which case the minimum time spent in saving states of the system in the worse case is $S_{ij} + f_j\ (L_{ij} + t_j)$. The optimal decision $x_{ij}(r)$ and the time spent in saving states of the system in the worse case after task i and if task j is processed next are clearly given by equations (1) and (2).

Since $f_i(r) = \max\{g_{ij}(r)\}$ it follows that $f_i(r)$ is the minimum time spent in saving states of the system after task i in the worst possible case. This completes the proof of the theorem. Two examples, figures 5 and 6, have been worked out.

Minimal Expected Save-Time Design

Let p_{ij} be the probability that task j is followed by task i. Let t_i, L_{ij}

and S_{ij} be expected rather than maximum values. Let us redefine $f_i(r)$ as

$$f_i(r) = \sum_j p_{ij} \, g_{ij}(r) \qquad (3a)$$

The algorithm used in the worst case design will determine the optimal decision rules for the minimal expected time design with eq. (3) replaced by (3a). The proof that the algorithm yields the optimal decision rules is similar to the proof of theorem 1 and is not presented here.

Computation

The amount of computation is proportional to the sum of the number of vertices and edges in the graph. The computation of $f_i(r)$ and $g_{ij}(r)$ are straightforward, since all functions are of the form:

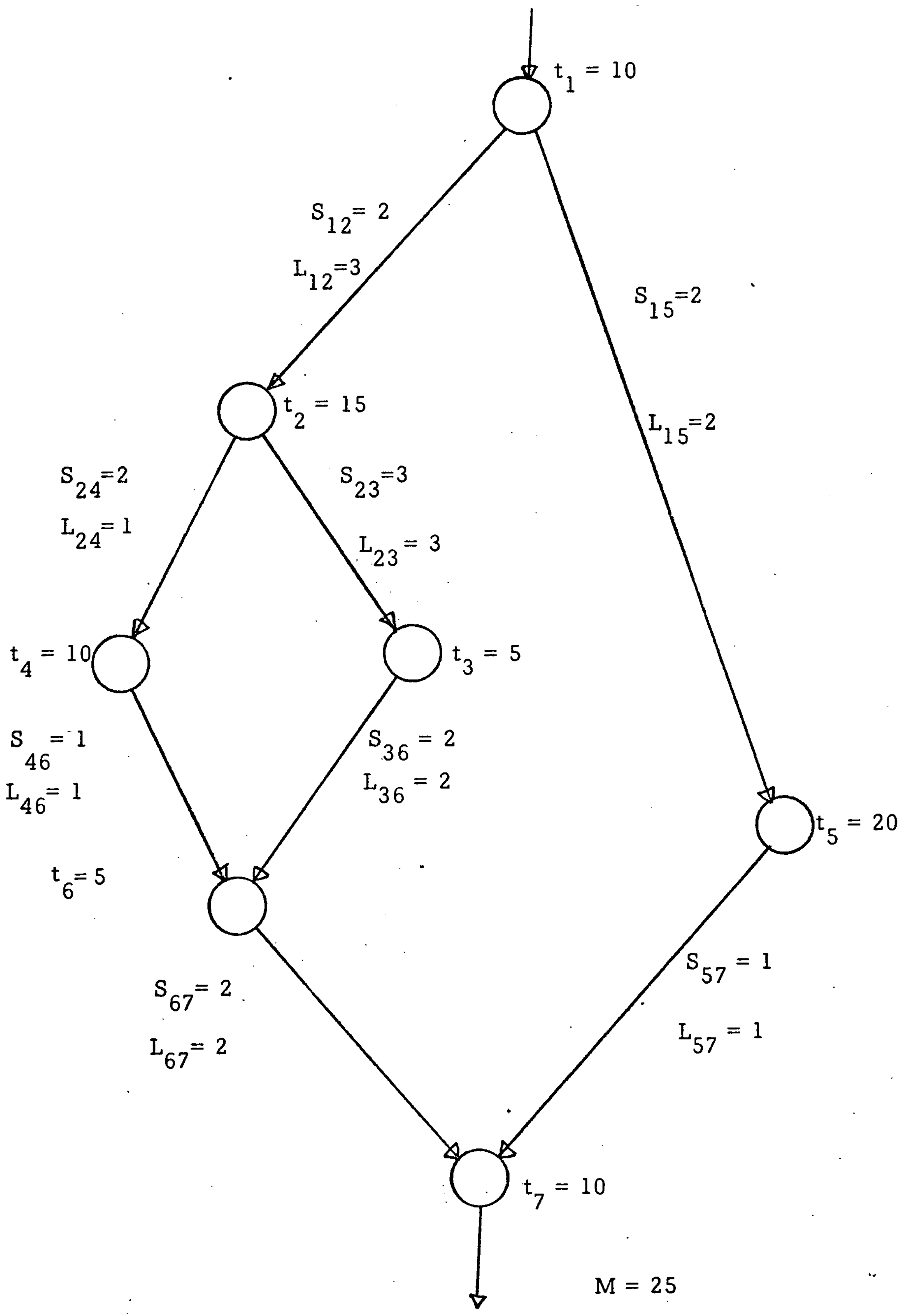
K_p for $q_p < r \leq q_{p+1}$, $p = 1, 2, \ldots T \ldots$, where the K_p are constants.

If all the data, t_i, L_{ij}, S_{ij}, M, in the problem are integers, then clearly q_p, $p = 1, \ldots, T$ are also integers. Hence the maximum number of discontinuities T, in the functions $f_i(r)$ and $g_{ij}(r)$, cannot exceed M.

The computation is most efficiently carried out by means of lists. The list structures and list processing techniques used are described later.

EXAMPLE 1

FIGURE 5



SOLUTION TO EXAMPLE 1

Initialization (0th Step)

There is only one exit vertex (i.e., a vertex without successors), viz. vertex 7.

Define $f_7(r) = \begin{cases} 0 \text{ for } 0 < r \leq 25 \\ \infty \text{ for } 25 < r \end{cases}$

Label vertex 7 (with a check ✓) to show that the f function for vertex 7 has been determined.

1st Step At this stage we note that only vertex 7 has been labeled. We note that vertices 5 and 6 have all their successors labelled.

Compute $g_{67}(r)$, $x_{67}(r)$ and B_{67} from equations (1) and (2).

$$g_{67}(r) = \begin{array}{l} S_{67} + f_7\ (L_{67} + t_7) \text{ if } r + t_7 > M \\ \min\ \{f_7\ (r + t_7),\ S_{67} + f_7\ (L_{67} + t_7)\} \text{ if } r + t_7 \leq M \end{array}$$

$$S_{67} + f_7\ (L_{67} + t_7) = 2 + f_7\ (2 + 10) = 2 + f_7\ (12) = 2$$

$$f_7\ (r + t_7) = \begin{cases} 0 \text{ for } 0 < r \leq 15 \\ \infty \text{ for } 15 < r \end{cases}$$

Hence

$$g_{67}(r) = \begin{cases} 0 \text{ for } 0 < r \leq 15 \\ 2 \text{ for } 15 < r \leq 25 \end{cases}$$

Compute $x_{67}(r)$ from

$$x_{67}(r) = \begin{cases} 0 \text{ if } g_{67}(r) = f_7\ (r + t_7) \\ 1 \text{ otherwise} \end{cases}$$

Hence $x_{67}(r) = \begin{cases} 0 \text{ for } 0 < r \leq 15 \\ 1 \text{ for } 15 < r \leq 25 \end{cases}$

and $B_{67} = 15$

We next compute $f_6(r)$ from:

$f_6(r)$ = maximum over all edges $(6,j)$ of $\{g_{6j}\ .(r)\ \}$

Since there is only one edge (6,7) leaving vertex 6, we have

$$f_6(r) = g_{67}(r) \text{ for } r \le 25$$

Hence $f_6(r) = \begin{cases} 0 & \text{for } 0<r\le 15 \\ 2 & \text{for } 15<r\le 25 \\ \infty & \text{for } 25<r \end{cases}$

Label vertex 6 to show that $f_6(r)$ has been determined.

2nd Step

Now we compute $g_{57}(r)$, $x_{57}(r)$, B_{57} from equations (1) and (2), since vertex 5 has all of its successors labelled.

$$g_{57}(r) = \begin{cases} S_{57} + f_7\,(L_{57} + t_7) & \text{if } r + t_7 > M \\ \min\,\{f_7(r+t_7),\ S_{57} + f_7\,(L_{57} + t_7)\} & \text{if } r + t_7 \le M \end{cases}$$

$$S_{57} + f_7\,(L_{57} + t_7) = 1 + f_7\,(1 + 10) = 1 + f_7\,(11) = 1$$

$$f_7\,(r + t_7) = \begin{cases} 0 & \text{for } 0<r\le 15 \\ \infty & \text{for } 15<r \end{cases}$$

Hence $g_{57}(r) = \begin{cases} 0 & \text{for } 0<r\le 15 \\ 1 & \text{for } 15<r\le 25 \end{cases}$

$x_{57}(r) = \begin{cases} 0 & \text{for } 0<r\le 15 \\ 1 & \text{for } 15<r\le 25 \end{cases}$

$B_{57} = 15$

We next compute $f_5(r)$ from

$$f_5(r) = \text{max over all edges } (5,j) \text{ of } \{g_{5j}\,.\,(r)\}$$

Since there is only one edge (5, 7) out of vertex 5, we have $f_5(r) = g_{57}(r)$.

Hence $f_5(r) = \begin{cases} 0 & \text{for } 0<r\le 15 \\ 1 & \text{for } 15<r\le 25 \\ \infty & \text{for } 25<r \end{cases}$

Label vertex 5 to show that $f_5(r)$ has been computed.

3rd Step

At this point, vertices 5, 6, and 7 have been labeled. We note that vertices 3 and 4 have all their successors labeled.

We compute $g_{36}(r)$, $x_{36}(r)$ and B_{36}.

$$S_{36} + f_6(L_{36}+t_6) = 2 + f_6(2 + 5) = 2 + f_6(7) = 2$$

$$f_6(r+t_6) = \begin{cases} 0 & \text{for } 0<r\le 10 \\ 2 & \text{for } 10<r\le 20 \\ \infty & \text{for } 20<r \end{cases}$$

Hence $g_{36}(r) = \begin{cases} 0 & \text{for } 0<r\le 10 \\ 2 & \text{for } 10<r\le 25 \end{cases}$

Since $g_{36}(r) = f_6(r+t_6)$ for $r \le 20$,
we have $x_{36}(r) = \begin{cases} 0 & \text{for } 0<r\le 20 \\ 1 & \text{for } 20<r\le 25 \end{cases}$

and $B_{36} = 20$

We now compute $f_3(r)$ and since there is only one edge going out of node 3, namely edge (3,6), we get $f_3(r)$ to be the same as $g_{36}(r)$.

Hence $f_3(r) = \begin{cases} 0 & \text{for } 0<r\le 10 \\ 2 & \text{for } 10<r\le 25 \\ \infty & \text{for } 25<r \end{cases}$

Label vertex 3 to show that $f_3(r)$ has been determined.

4th Step.

We note that vertex 4 has all of its successors labelled. Hence we similarly compute $g_{46}(4)$, $x_{46}(r)$, B_{46} and $f_4(r)$

$$g_{46}(r) = \begin{cases} 0 & \text{for } 0<r\le 10 \\ 1 & \text{for } 10<r\le 25 \end{cases}$$

$$x_{46}(r) = \begin{cases} 0 & \text{for } 0<r\le 10 \\ 1 & \text{for } 10<r\le 25 \end{cases}$$

$$B_{46} = 10$$

$$f_4(r) = \begin{cases} 0 & \text{for } 0 < r \leq 10 \\ 1 & \text{for } 10 < r \leq 25 \\ \infty & \text{for } 25 < r \end{cases}$$

5th Step

At this stage vertices 3,4,5,6 and 7 have been labeled. We note that vertex 2 has all of its successors labelled. We now compute $g_{23}(r)$, $x_{23}(r)$, B_{23}.

$$S_{23} + f_3 \ (L_{23} + t_3) = 3 + f_3 \ (3 + 5) = 3 + f_3(8) = 3$$

$$f_3(r + t_3) = \begin{cases} 0 & \text{for } 0 < r \leq 5 \\ 2 & \text{for } 5 < r \leq 20 \\ \infty & \text{for } 20 < r \end{cases}$$

Hence $g_{23}(r) = \begin{cases} 0 & \text{for } 0 < r \leq 5 \\ 2 & \text{for } 5 < r \leq 20 \\ 3 & \text{for } 20 < r \leq 25 \end{cases}$

$$x_{23}(r) = \begin{cases} 0 & \text{for } 0 < \ 0 < r \leq 20 \\ 1 & \text{for } 20 < r \leq 25 \end{cases}$$

$$B_{23} = 20$$

Similarly, we get

$$g_{24}(4) = \begin{cases} 1 & \text{for } 0 < r \leq 15 \\ 3 & \text{for } 15 < r \leq 25 \end{cases}$$

$$x_{24}(r) = \begin{cases} 0 & \text{for } 0 < r \leq 15 \\ 1 & \text{for } 15 < r \leq 25 \end{cases}$$

$$B_{24} = 15$$

Hence $f_2(r) = \max \{g_{23}(r), g_{24}(r)\}$

$$= \begin{cases} 1 \text{ for } 0 < r \le 5 \\ 2 \text{ for } 5 < r \le 15 \\ 3 \text{ for } 15 < r \le 25 \end{cases}$$

We label vertex 2 to show that $f_2(r)$ has been computed.

<u>6th Step</u>

At this stage vertices 2,3,4,5,6,7 have been labelled. We note that vertex 1 has all of its successors labeled.

Hence we compute $g_{12}(r)$, $x_{12}(r)$, B_{12} and $g_{15}(r)$, $x_{15}(r)$, B_{15}.

$$g_{12}(r) = \begin{cases} 3 \text{ for } 0 < r \le 10 \\ 5 \text{ for } 10 < r \le 25 \end{cases}$$

$$x_{12}(r) = \begin{cases} 0 \text{ for } 0 < r \le 10 \\ 1 \text{ for } 10 < r \le 25 \end{cases}$$

$$B_{12} = \{10$$

$$g_{15}(r) = \begin{cases} 1 \text{ for } 0 < r \le 5 \\ 3 \text{ for } 5 < r \le 25 \end{cases}$$

$$x_{15}(r) = \begin{cases} 0 \text{ for } 0 < r \le 5 \\ 1 \text{ for } 5 < r \le 25 \end{cases}$$

$$B_{15} = 5$$

Hence $f_1(r) = \max \{g_{12}(r), g_{15}(r)\}$

$$= \begin{cases} 3 \text{ for } 0 < r \le 10 \\ 5 \text{ for } 10 < r \le 25 \\ \infty \text{ for } 25 < r \end{cases}$$

Label vertex 1.

7th Step

Since all the vertices have been labelled the algorithm terminates.

To summarize the solution to the problem:

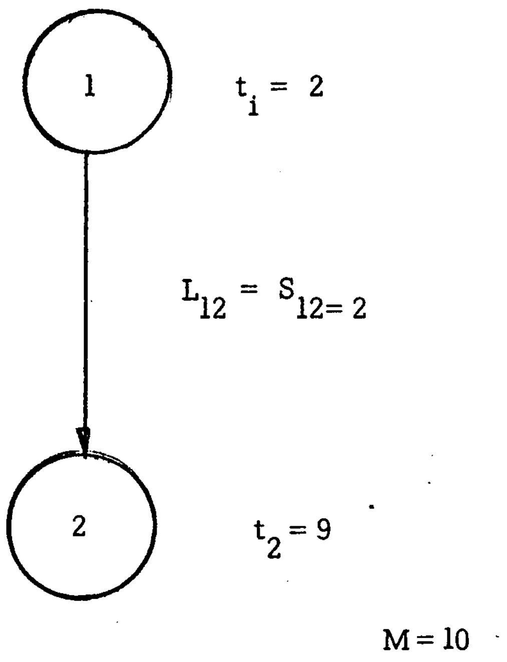
The breakpoints are: $B_{12}=10$, $B_{15}=5$, $B_{23}=20$, $B_{24}=15$, $B_{46}=10$

$B_{36}=20$, $B_{67}=15$, $B_{57}=15$

Let the time L_o taken to load the program initially be 1 unit. Then the time spent in saving system environments in the worst case is $f_i(L_o + t_i) = f_i(11) = 5$ units. However, the insertion of rollback points will be dynamic as shown below.

Suppose that in a given run of the above program tasks 1,2,4,6 and 7 are executed in sequence, and suppose all tasks take 3 units of time. Then no rollback points will be inserted. If on another run of the same program, the same tasks, 1,2,4,6 and 7 are executed in sequence, and all tasks that take 5 units of time, a single rollback point will be inserted on edge (4,6); i.e. the system environment will be saved after task 4 is complete and before task 6 is initiated. If on yet another run of the same program, the same tasks are executed, and each task takes its maximum time, i.e. task 1 takes 10 units of time, task 2 takes 15 units of time etc., rollback points will be inserted between tasks 1 and 2, 2 and 4, 4 and 6, but not between 6 and 7. Hence, we see, that real-time decisions are made as to which edges are to contain rollback points; these decisions are clearly a function of input data. However, even though the insertion of rollback points is dynamic, the implementation of the algorithm is very simple, and requires negligible overhead, once the breakpoints B_{ij} are determined. Note that the breakpoints B_{ij} themselves are not a function of input data.

Example 2 Figure 6



Solution to Example 2, Figure 6

0th Step Vertex 2 is an exit vertex.

$$\text{Put} \quad f_2(r) = \begin{cases} 0 & \text{for } 0 < r \le 10 \\ \infty & \text{for } 10 < r \end{cases}$$

1st Step Since vertex 1 has all of its successors labelled, we find $g_{12}(r)$ and $f_1(r)$.

$$f_2(r + t_2) = \begin{cases} 0 & \text{for } 0 < r \le 1 \\ \infty & \text{for } 1 < r \end{cases}$$

$$S_{12} + f_2\,(L_{12} + t_2) = 2 + f_2(2 + 9) = 2 + f_2(11) = \infty$$

$$\text{Hence } g_{12}(r) = \begin{cases} 0 & \text{for } 0 < r \le 1 \\ \infty & \text{for } 1 < r \end{cases}$$

Clearly $f_1(r) = g_{12}(r)$, since there is only one edge out of vertex 1.

Let the time L_0 taken to load the program initially be 1 unit. Then, since $f_1\,(L_0 + t_1) = f_1(1 + 2) = \infty$, there exists no feasible solution to the problem.

The Code for the Algorithm.

A graphical example for computing $g_{ij}(r)$ is shown in Figure 7, and of $f_i(r)$ is shown in Figure 8.

Consider the problem shown in Figure 9. The input data is of the form shown in table 1. The first row, for instance, of table 1 states that task 1 may be succeeded by task 2, and $L_{12} = 1$, $S_{12} = 1$.

Three linked lists are used for storing information. They are called the NODE, EDGE and FUNCTION lists. Each cell in the NODE list has five fields (table 2). Cell (row) 3 of table 2 states that vertex 3 has 1 (content of COUNT field) successor; the list of predecessors of vertex 3 starts in cell number 2 (content of TOP field) of the EDGE list; the list of successors starts at cell number 4 (content of BOTTOM field) of the EDGE list; ←

the estimated maximum time of task 3 is 10 units; FLINK points to the cell in the FUNCTION list where the first element of the $f_3(r)$ function is stored.

Each cell in the EDGE list has seven fields (table 3). PREDECESSOR NODE and PREDECESSOR LINK fields are used to keep a linked list of the predecessors of a node. SUCCESSOR NODE and SUCCESSOR LINK fields are used to keep a linked list of the successors of a node. For instance the first cell in the list of predecessors of vertex 3 is cell number 2 of the EDGE list. The PREDECESSOR NODE field of cell number 2 of the EDGE list is 2, since vertex 2 is a predecessor of vertex 3. The PREDECESSOR LINK field links the list of predecessors. The LOAD and SAVE TIME fields are self explanatory, the load time of 3 and the save time of 3 in the second cell of the EDGE list refer to edge (2,3). GLINK points to a cell in the FUNCTION list where the first element of the $g_{23}(r)$ function is stored.

The f ( ) and g ( ) functions are step functions. They are stored by storing the breakpoints. For instance, a step function and the method of storing it as a linked list are shown in Figure 10. The f and g functions are stored as linked lists; each cell in the FUNCTION list has three fields: X, Y, and XYLINK.

Note that a task graph with M nodes and N edges needs only N+M cells for the NODE and EDGE list (A graph with 1000 nodes and 3000 edges needs only 4000 cells). The size of the FUNCTION LIST varies; the f and g functions are computed when required and the storage occupied by the f and g functions are returned to free storage when they are no longer required.

This method of storing the f and g functions is economical and allows for easy computation. For instance, evaluating $f_i(r)$ from $f_i(r) = \max \{ g_{ij_1}(r), \ldots, g_{ij_p}(r) \}$ can be done readily by merely inspecting the points of discontinuity (i.e. the contents of the X and Y fields of the cells) of $g_{ij_1}(r), \ldots, g_{ij_p}(r)$.

Experimental Results

The algorithm has been coded in FORTRAN and run on a CDC 6600. Thirty problems were generated using the random number generator on the CDC 6600. Each problem had roughly 200 nodes and 400 edges. All problems were solved in less than 0.11 seconds. The computational results are shown in table 4.

Conclusions:

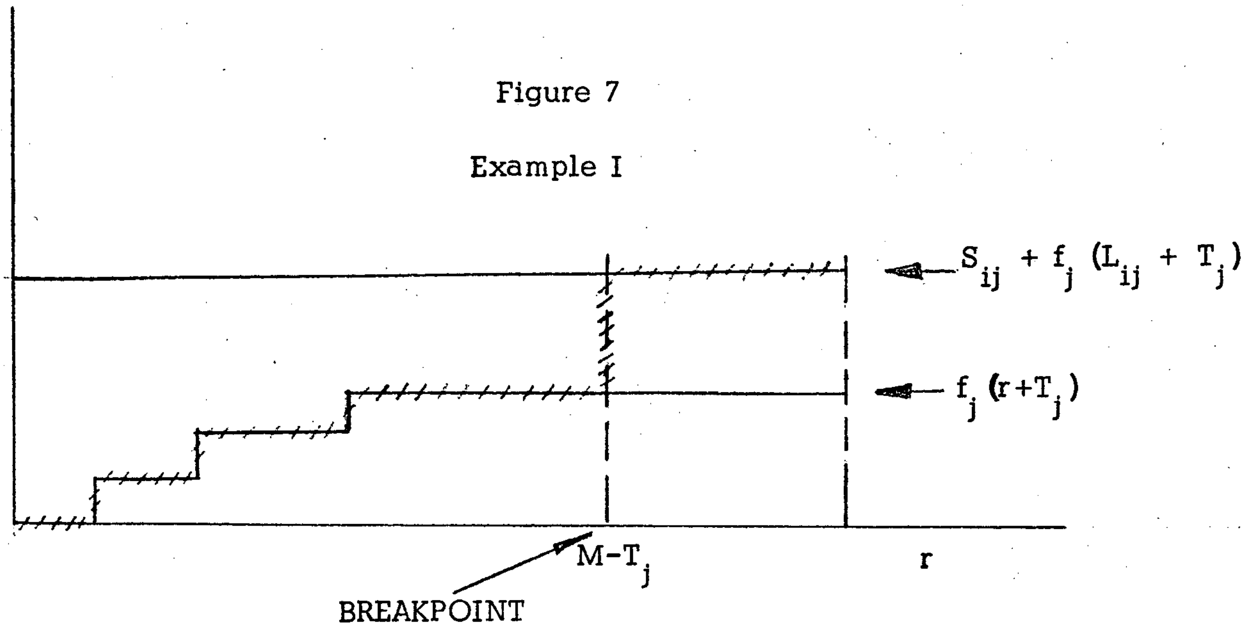
The rollback problem has been described. Different models for the rollback problem have been compared and an optimal algorithm for one of the models has been presented. The list structures used in coding the algorithm have been discussed. Some experimental results obtained by running the code on a CDC 6600 have been presented.

The model has two possible drawbacks. Firstly, it is hard to accurately estimate the maximum execution times of tasks. Secondly, an accurate description of a program may require that the program graph have a very large number of nodes. The second drawback is vitiated since the algorithm is efficient and does not require much storage or processing time to analyze large graphs.

As in many modeling problems these days, the major "cost" of using the model is the time required to obtain data and estimates rather than the time required to run the algorithm on a computer. Improvements should primarily be concerned with models using less data and fewer estimates.

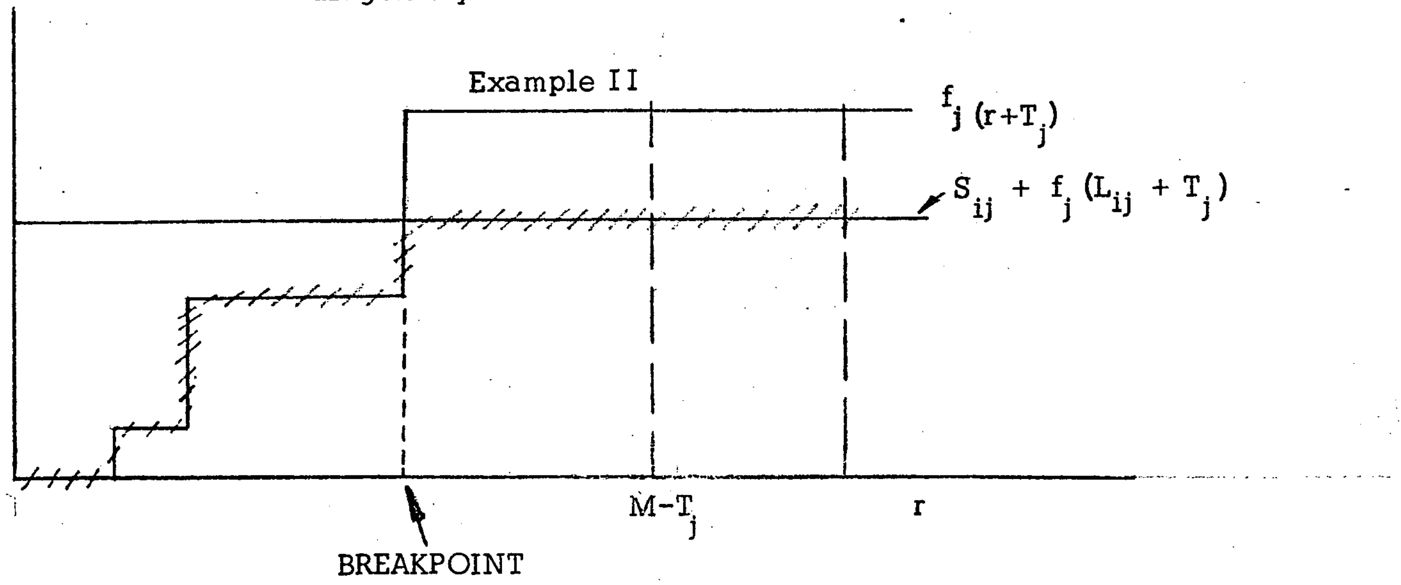
Figure 7

Example I



The $g_{ij}(r)$ function is marked by diagonally hatched lines.

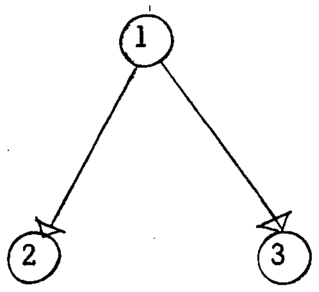
Example II

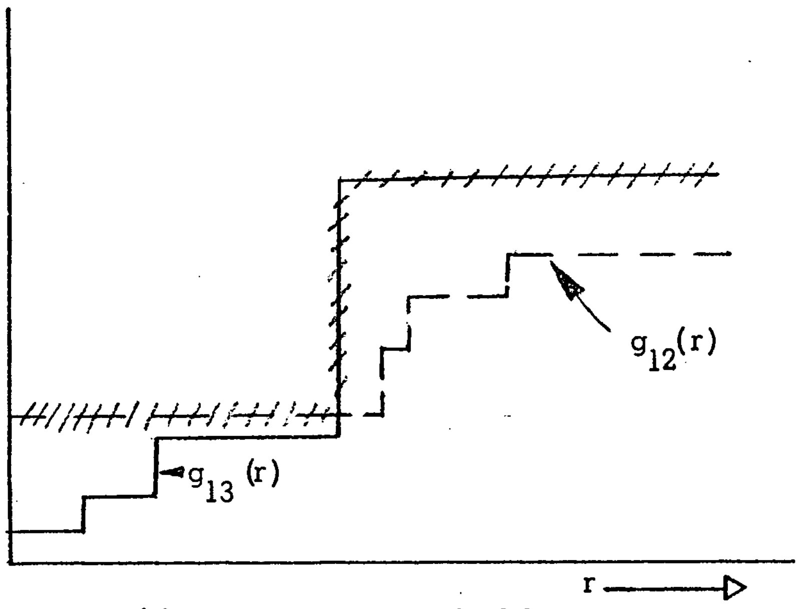


Store the system environment immediately after task i is complete and if task j is called next if and only if r > BREAKPOINT, where r is the recovery time when task i is complete.

Figure 8

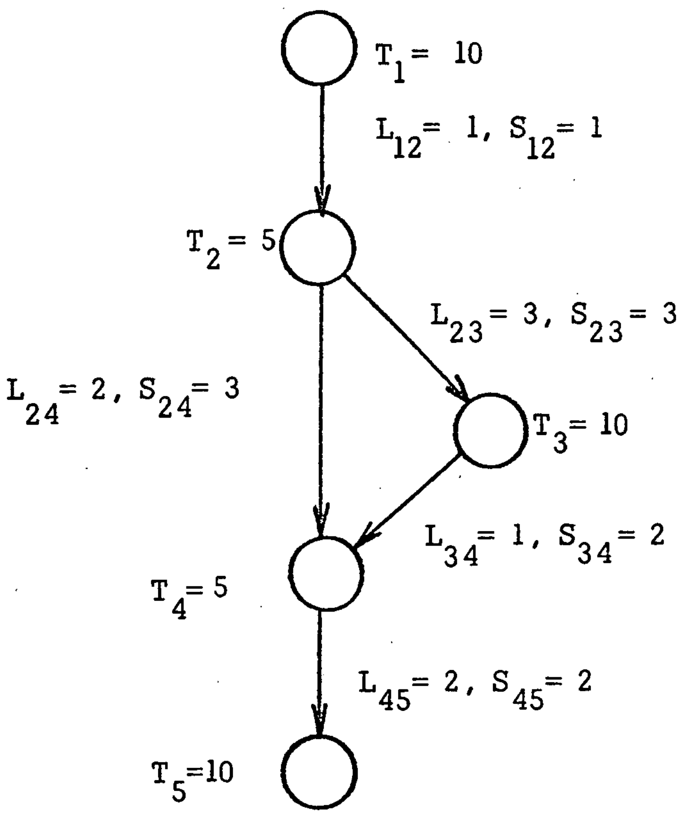
Computing $f_1(r)$ Given $g_{12}(r)$ and $g_{13}(r)$





$f_1(r)$ is shown in hatched lines.

Figure 9



Successor Task Number	Predecessor Task Number	Load Time	Save Time
2	1	1	1
3	2	3	3
4	2	2	3
4	3	1	2
5	4	2	2

Table 1

{The number of rows in the table. {= The number of edges in the graph.

NODE LIST

	Count	Top	Bottom	Time	Flink
Cell No. 1	1	0	1	10	0
Cell No. 2	2	1	3	5	0
Cell No. 3	1	2	4	10	0
Cell No. 4	1	4	5	5	0
Cell No. 5	0	5	0	10	0

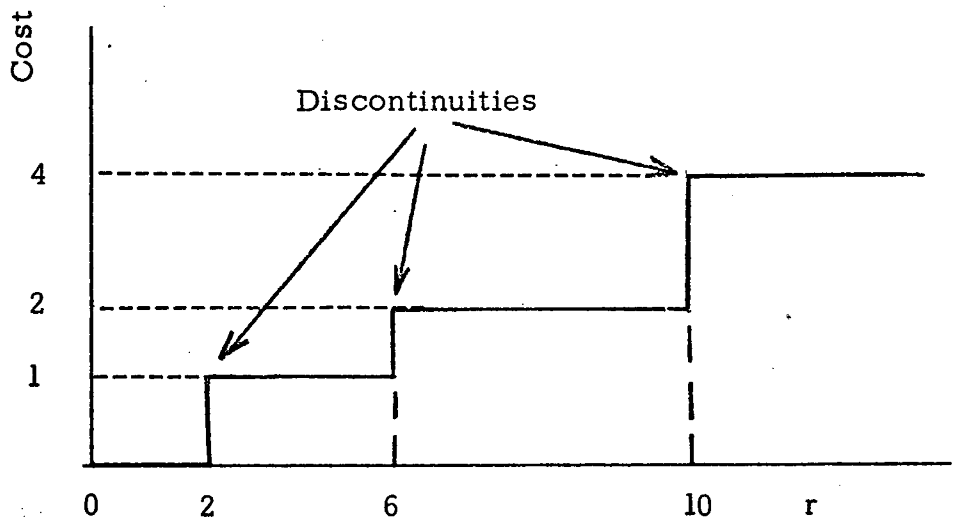
Table 2

EDGE LIST

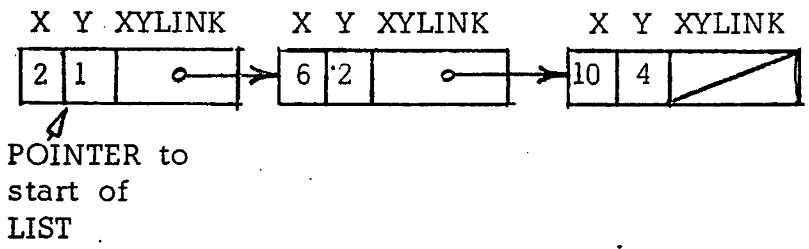
	Predecessor Node	Predecessor Link	Successor Node	Successor Link	Load Time	Save Time	Glink
Cell No. 1	1	0	2	0	1	1	0
Cell No. 2	2	0	3	0	3	3	0
Cell No. 3	2	0	4	2	2	3	0
Cell No. 4	3	3	4	0	1	2	0
Cell No. 5	4	0	5	0	2	2	0

Table 3

Figure 10



X (abcissa) of discontinuity	2	6	10
Y (ordinate) of discontinuity	1	2	4



FUNCTION LIST

	X	Y	XYLINK
Cell Number 1	2	1	3
Cell Number 2	1	1	0
Cell Number 3	6	2	4
Cell Number 4	10	4	0

TABLE 4

PROBLEM NUMBER	1	2	3	4	5	6	7	8	9	10
NUMBER OF NODES:	220	225	176	200	230	233	182	188	222	244
NUMBER OF EDGES:	429	419	318	397	469	446	361	370	434	447
EXECUTION TIME : IN SECONDS	.099	.087	.068	.093	.104	.095	.076	.085	.095	.098

PROBLEM NUMBER:	11	12	13	14	15	16	17	18	19	20
NUMBER OF NODES :	223	201	192	226	212	217	200	216	192	230
NUMBER OF EDGES :	429	388	382	458	410	424	380	417	377	434
EXECUTION TIME : IN SECONDS	.098	.087	.086	.096	.093	.096	.077	.091	.086	.098

PROBLEM NUMBER:	21	22	23	24	25	26	27	28	29	30
NUMBER OF NODES:	220	206	161	229	217	215	192	254	175	179
NUMBER OF EDGES:	456	400	302	449	431	421	361	480	350	325
EXECUTION TIME IN: SECONDS	.102	.078	.067	.105	.103	.088	.084	.110	.083	.073

REFERENCES

1. G. Oppenheimer, K. P. Clancy. Considerations of software protection and recovery from hardware failures. Proc. FJCC 1968.

2. A. N. Higgins. Error recovery through programming. Proc. FJCC 1968.

3. R. E. Bellman, S. E. Dreyfus. Applied dynamic programming. Princeton University Press 1962.

4. J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components.

   Automata Studies, Annals of math. studies no. 34. (C. E. Shannon and J. McCarthy eds.) Princeton University Press 1956.

5. J. Martin. Design of real-time computer systems. Prentice-Hall Englewood Cliffs N. J. 1967.

6. A. Cowan. Software and hardware reliability. Forthcoming M. S. thesis in computer sciences at the University of Texas at Austin.

7. W. H. Pierce. Failure tolerant computer design. Academic Press, New York 1965.

8. F. P. Matur, A. Avizienis. Reliability analysis of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair. Proc. SJCC 1970.

9. D. N. Freeman. A storage-hierarchy system for batch-processing. Proc. SJCC 1968.

10. C. V. Ramamoorthy. A structural theory of machine diagnosis. Proc. SJCC 1967.

11. C. V. Ramamoorthy, K. M. Chandy, M. J. Gonzalez. Optimal Scheduling Strategies in a Multiprocessor System. To appear in IEEE Trans. on EC.

12. E. C. Russell, G. Estrin. Measurement based automatic analysis of FORTRAN programs. Proc. SJCC 1969.

13. B. Beizer. Analytical techniques for the statistical evaluation of program running time. Proc. FJCC 1970.

14. R. C. Daley, P. G. Neumann. A general purpose file system for secondary storage. Proc. FJCC 1965.